Code Generation for DHAD on VAX

by

Yahya Mohammad Saleh Garout

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

June, 1989

Order Number 1355747

Code generation for DHAD on VAX

Garout, Yahya Mohammad Saleh, M.S.

King Fahd University of Petroleum and Minerals (Saudi Arabia), 1989

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

CODE GENERATION FOR DHAD ON VAX

BY

YAHYA MOHAMMAD SALEH GAROUT

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
In
COMPUTER SCIENCE

JUNE 1989

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN 31261, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by YAHYA MOHAMMAD SALEH GAROUT under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate studies, in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE in COMPUTER SCIENCE.

THESIS COMMITTEE

Thesis Advisor

Member

Member

Department Chairman

Dean, College of Graduate Studies

14-3-93

Date

To My Parents, Brothers, and Sisters

ACKNOWLEDGMENT

Acknowledgment is due to King Fahd University of Petroleum and Minerals for providing the opportunity to carry out this research work.

I would like to express my deep appreciation to my thesis advisor ***Dr. Mohammed Ghazaly khayat*** for his patient guidance, tremendous help, and encouragement.

I would also like to express my thanks to the other members of my thesis Committee ***Dr. Mustapha Ziad*** and ***Dr. Manzer Masud*** for their helpful comments.

Special thanks are due to ***my brother Khalid*** for his concern and encouragement,to ***Ayman Nazzal, Siyad Ma, Ahmad Ghazal, Anan Yaagoub*** for their assistance, to Khalid Hamied, Khalid Al-tawil, Abdul-hakim Nazzal, Ayman Al-melh, Khalid Al-melh, Adel Lafi, Yahya Zaidan, Azzam Ibrahim,Husni Al-muhtaseb,Jaweed Yazdani, Majed Samhan, Nadir Abdul-Hadi, Ahmad Awad, Mohammad Melhi, and all my colleagues.

خلاصة الرسالة

اسم الطالـــب : يحيى محمد صالح خليل قاروط .

عنوان الرسالـة : مولد تعليمات حاسبات الفاكس للغة الضاد .
التخصـــــص : علوم وهندسة الحاسب الآلي .
تاريخ الشهـادة : يونيو ١٩٨٩م .

مولد تعليمات حاسبات الفاكس للغة الضاد

لقد أدى ازدياد استعمال اللغة العربية فى مجال الحاسبات إلى تطوير العديد من لغات البرمجة العربية . والضاد هي واحدة من لغات البرمجة العربية والتي تعد ضمن لغات البرمجة الهيكلية مثل لغة الباسكال . ومن الضروري نقل مترجم الضاد إلى الحاسبات ذات الانتشار للإستفادة منه في تطوير البرامج العربية والبحوث والتعليم .

إن الهدف الرئيسي لهذا البحث هو إعادة كتابة مولد تعليمات حاسبات الفاكس للغة الضاد . ومولد التعليمات هو الجزء الذي يقوم بتوليد تعليمات لغة التجميع من لغة متوسطة المستوى ينتجها المترجم في المراحل السابقة . وقد تضمن البحث دراسة مولد التعليمات الذي ينتج لغة التجميع الخاصة بوحدة المعالجة (زد- ٨٠). وتم استخدام نفس الخوارزميات مع بعض التعديلات . ولقد تم توليد التعليمات بلغة التجميع الخاصه بالفاكس لجميع عمليات نماذج المعلومات (الوحدات ، الأعداد الصحيحة ، الأعداد الحقيقية ، القيم المنطقية ، الحروف ، السلاسل الرمزية ، السجلات ، المصفوفات ، المؤشرات ، الملفات) ، بالاضافة إلى توليد العمليات لتحضير عناوين المتغيرات والانتقال والإدخال والإخراج واستدعاء البرامج الفرعية والدالات . ولقد طُوّر مولد التعليمات باستخدام لغة الباسكال وكان حجم البرنامج القابل للتنفيذ حوالي ٤٠ ألف حرف .

درجة الماجستير في العلوم
جامعة الملك فهد للبترول والمعادن
الظهران - المملكة العربية السعودية
يونيو ١٩٨٩ م

THESIS ABSTRACT

NAME OF STUDENT: YAHYA MOHAMMAD SALEH GAROUT

TITLE OF STUDY : CODE GENERATION FOR DHAD ON VAX

MAJOR FIELD : COMPUTER SCIENCE AND ENGINEERING

DATE OF DEGREE : JUNE 1989

The increasing use of Arabic in computers has led to the development of Arabic programming languages. DHAD is an Arabic programming language which falls into the class of structured languages such as Pascal. Porting the DHAD compiler to popular computers (such as the VAX family of computers) is needed to make the language available for Arabic software development, research, and education.

The main objective of this research is to rewrite the code generator module of the DHAD compiler for the VAX computers. The final code generator is the module of the compiler that produces assembly language code from the intermediate file taken as input. The work included a study of the final code generator developed for a Z-80 based computer. The same algorithm for code generation was used with some modifications. Code generated for the operations on the various data structures (byte, word, integer, real, logical, string, records, arrays, pointers and files) supported by DHAD has been produced in VAX assembly language. Code for operations, including address preparation of operands, branching, I/O operations, standard and user-defined function and procedure call, definition and parameter preparation, has also been implemented in VAX assembly language. Run-time support library routines were used for standard functions and procedures. In addition, some minor modifications of the other modules (lexical and syntax analyzers) of the compiler were imposed by the transportation of DHAD to the VAX. The final code generator was implemented using Pascal on the VAX computer. The size of the executable code of the code generation module was about 40 Kbytes.

MASTER OF SCIENCE DEGREE

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

Dhahran, Saudi Arabia

June 1989

TABLE OF CONTENTS

Chapter I

INTRODUCTION

1.1 Introduction

The increasing use of Arabic in computer applications has led to the development of many Arabic programming languages. Unlike most Arabic programming languages which are Basic-like, DHAD falls into the class of structured programming languages which includes Pascal and C. Porting the DHAD compiler to different computer systems is required to make the language available for Arabic software development, research, and education.

1.2 Scope of Work

The main objective of this thesis is to rewrite the final code generator module of the DHAD compiler in order to produce assembly code for the VAX machines running under the VMS operating system.

The work is to also include a study of the Z-80 final code generator, the architecture and the assembly language of the VAX machines and a review of the literature on code

generators.

1.3 Thesis Organization

This thesis is organized as follows: chapter II reviews the history and features of DHAD. In chapter III, the VAX architecture and assembly language are briefly introduced. An overview of code generators is discussed in chapter IV. Chapter V explains the implementation details of the final code generator on the VAX computers. Conclusions and future work are presented in chapter VI.

Chapter II

HISTORY AND FEATURES OF DHAD

2.1 Introduction

DHAD is an Arabic General-purpose structured programming language[13] . Unlike other Arabic programming languages, which are Basic-like, DHAD is Pascal-like. It supports user-defined data types, structured data types, control constructs, I/O formatting capability, and recursion. The compiler was implemented using Pascal on a Z-80 based micro-computer system as a research project supported by KACST[13]. In this chapter, we present the basic constructs of DHAD, the structure of DHAD programs, and the structure of the DHAD compiler.

2.2 Language Constructs

The basic constructs can be divided into three classes : data structures, control structures, and I/O facilities.

2.2.1 The Data Structures

The data structures of DHAD include characters, integers, reals, logical, pointers, scalars, arrays, records, files, and sets. The operations allowed on these data types are the same as those in Pascal. In addition, DHAD supports : byte, word, string, list, queue, stack, and binary tree types. The operations that can be performed on bytes and words are arithmetic, logical, comparison, assignment and I/O operations. Compatibility between bytes and characters, bytes and words, bytes and integers, bytes and Booleans, and words and integers is also supported. Operations on strings include comparisons, assignment and accessing of string elements. The list operations are inserting and deleting an element, testing for equality and non-equality, and the emptiness test. The operations on queues and stacks are the same as those of lists. Testing for emptiness, extracting the root, and the right and left subtrees, and inserting and deleting right and left subtrees are the operations permitted on binary trees.

2.2.2 The Control structures

DHAD control structures include the assignment statement, selection, repetition and unconditional transfer constructs.

The assignment statement is similar to that of Pascal. For selection, there is only one construct which is similar to the CASE structure in Pascal. Its label can be an expression while in Pascal it must be a constant. There are four repetition constructs in DHAD which are equivalent to Pascal's WHILE, REPEAT-UNTIL, REPEAT-FOREVER and FOR loops. The REPEAT-FOREVER loop can be exited by a QUIT statement. The scope of a loop must be enclosed by a pair of matching parentheses. There are ten pairs of compound parentheses: (<* , *>), (<# , #>), (<! , !>), (<. , .>), (<: , :>), (/* , */) , (/# , #/), (* , *\ ), (\# , #\ ) and (%* , *%). These parentheses are equivalent to BEGIN and END in Pascal; however, they are better for debugging and reading programs. The unconditional transfer constructs of DHAD include the equivalents of GOTO, CALL, QUIT, STOP and RETURN statements. QUIT terminates loops, while STOP is used to stop the execution of the program no matter where it is. More than one statement can be written on the same line separated by blanks which simplifies the syntax of DHAD and provides flexibility.

2.2.3 Input and Output Facilities

DHAD supports constructs for formatted and unformatted I/O

operations on both sequential and random access files. The formatting facilities of DHAD are similar to those of Fortran.

2.3 The Structure of DHAD Programs

A DHAD program consists of a main program followed by the bodies of subprograms (functions and procedures). Procedure recursion is supported, while function recursion is not. Procedures and functions must be declared in the main program or any subprogram. Each program or subprogram consists of two sections: a declarations section and an operations section. In the declarations section, the order of declarations is: type, constant, variables, subprograms and labels respectively. Constants can be used in the declaration of types. The operations section consists of a sequence of operations to be carried out . Each of the declarations and the operations sections should be enclosed within a pair of compound parentheses. Comments must be on a separate line starting with a question mark.

2.4 The Structure of the DHAD Compiler

The DHAD compiler consists of three modules:

1. Lexical Analyzer,
2. Syntax and Semantics Analyzer, and
3. Final Code Generator.

We will now discuss each module briefly .

2.4.1 The Lexical Analyzer

This module consists of two submodules : the internal code generator and the token generator. The functions of the internal code generator [13] are:

a. transforming the source program into a sequence of codes using a code table for characters.
b. deleting blanks and inserting a special marker between words.
c. transforming compound parentheses and multi-character operators into single codes.
d. balancing both simple and compound parentheses.
e. producing a listing of the source program with proper line numbers and nesting level numbers.
f. deleting comment statements.
g. inserting an end of statement marker after each simple statement.

h. issuing proper error messages.

The token generator performs the following functions [13]:

a. producing code for reserved words, identifiers,and constants.
b. producing the corresponding values for tokens produced in the above step.
c. initializing symbol tables and internal constants.
d. issuing proper error messages.

The lexical analyzer reads the source program line by line. The output is two elements : token code and token value. The lexical analyzer also produces a listing with error messages. The lexical analyzer is called by the syntax analyzer to provide the next token.

2.4.2 The Syntax and Semantics Analyzer

This module consists of two submodules : the declarations section analyzer and the operations section analyzer. Both modules use the recursive descent method [13,16]. Each of these two submodules is invoked for every program or subprogram.

The functions of the declarations section analyzer are [13]:

a. checking the syntax of the declarations section.
b. checking the semantics of the declarations section.
c. building the symbol tables for user-defined types, constants, variables, subprograms and labels.
e. producing intermediate code to reserve memory space for structured constants and variables.
f. issuing proper error messages.

Three types of symbol tables are produced by this module. The first for types, constants and variables, the second for subprograms, and the third for labels.

The functions of the operations section analyzer are [13]:

a. checking the syntax of the operations section.
b. checking the semantics of the operations section.
c. generating intermediate code including code for non-declared structured constants.
d. issuing proper error messages.

The operations section module uses the postfix notation for algebraic expressions, and a modified form of triples for other operations.

The input to the syntax and semantics analyzer is the token codes and the token values produced by the lexical analyzer. The output of this module is a file containing the intermediate code.

2.4.3 The Final Code Generator

The functions of this module are [13]:

a. generating code to allocate space for variables and structured constants in the main program and in each subprogram.

b. generating final code for the intermediate file produced by the syntax and semantics analyzer.

c. optimizing the final code by storing intermediate results only when necessary, using immediate operands for simple constants and taking advantage of symmetric operations.

The input to this module is the intermediate file produced by the syntax and semantics analyzer. The output of this module is a file containing Z-80 assembly language code for the target machine. This module will be discussed in full detail in the fifth chapter.

Chapter III

VAX ARCHITECTURE AND ASSEMBLY LANGUAGE

3.1 Architecture

The VAX-11 series is a family of computer systems sharing common features such as the instruction set, instruction format, data storage elements, addressing modes, and addressing space. This family has the following features [4,12,14,17,18,19,20]:

- Virtual memory with an address space of 4 gigabytes. This memory is divided into two regions one for the system and the other for the users. The system uses the lower half of the memory and users have the higher half of the memory space.
- A data path width of 32 bits.
- An address width of 32 bits.
- A variable length instruction format.
- Microprogrammed CPU and a standard set of 300 instructions.

- Nine addressing modes which consist of :

  * register Rn

  in this mode Rn contains the operand.

  * register-deferred (Rn)

  in this mode Rn contains the address of the operand.

  * autodecrement -(Rn)

  the address of the operand is decremented by the length implied by the instruction. After that Rn contains the address of the operand.

  * autoincrement (Rn)+

  the address of the operand is in Rn then after fetching or storing the operand, the address in Rn is incremented by the length included in the instruction as a type.

  * autoincrement-deferred @(Rn)+

  in this mode Rn contains the address of the address of the operand and then Rn is incremented by four since addresses are four bytes long.

  * displacement D(Rn)

  in this mode the displacement is added to the contents of Rn which then points to the operand.

  * displacement-deferred @D(Rn)

  after the displacement is added to the contents of Rn, it forms the address of the address of the

operand.

* indexed. [Rn]

this addressing mode is normally used in conjuction with another addressing mode. The contents of Rn act as a relative index to the base address calculated from the other addressing mode associated with this index mode. All the previous addressing modes can be used with the index mode.

* literal #literal

here the value of the literal is the actual operand.

- Fourteen data types which are:
  - byte
  - word
  - longword
  - quadword
  - octaword
  - F_floating
  - D_floating
  - G_floating
  - H_floating
  - packed decimal string
  - character string
  - variable-length bit field
  - numeric string
  - queue.
- Sixteen general purpose registers each of 32 bits in length. Registers R0-R11 are general purpose, whereas R12-R15 are not really general purpose because they have special functions:
  - -R12 (or AP) is the argument pointer. Its function is that it contains the address of the argument list prepared whenever a procedure is called.
  - -R13 (or FP) is the frame pointer. Its function is pointing to the frame created when a procedure is called.

-R14 (or SP) is the stack pointer. Its function is that it points to the top of the user stack. When a procedure is called a frame for it is created; both SP and FP will point to the start of the frame. FP should not be changed by instructions, whereas SP will be changed with push and pop instructions.

-R15 (or PC) is the program counter and it contains the address of the next instruction to be fetched for execution.

Registers R0-R5 will hold certain information as a result of string operations, while R6-R11 are totally under the control of the user. R0 is used normally to hold a return code after a procedure call is finished.

3.2 Machine Instructions

The assembly language instructions can be classified into the following categories :

1. Data transfer instructions:

   This category of instructions moves data into a register, between registers, between a register and a memory location, or between memory locations. These operations include:

   - move.

- clear.
- move complemented.
- push longword.
- convert.
- move zero extended.
- move address.
- push address.

2. Arithmetic instructions:

   The arithmetic instructions perform the basic arithmetic operations on integers and reals, which include:

   - increment.
   - decrement.
   - add.
   - subtract.
   - multiply.
   - divide.

3. Logical instructions:

   The logical operations manipulate the bits of the operands such as:

   - bit set which implements the OR operation.
   - bit clear which implements the AND operation.
   - exclusive-OR.
   - bit test which checks the operand and then sets the

flags.

4. Comparison and test instructions:

   The compare instruction compares two operands and determines whether the first operand is less than, equal to, or greater than the second operand setting the flags accordingly. However the test instruction examines the contents of the operand and then sets the flag bits.

5. Branch instructions:

   The VAX assembly language contains a large set of branching instructions to cover unconditional, conditional, overflow and carry branches. The unconditional transfers contain three instructions: two for near jumps (branch using byte displacement and branch using word displacement) and the third for jumps with any displacement. In the conditional transfers, there are instructions, which use the flags, for jumps on:

   - less than.
   - less than or equal to.
   - equal to.
   - not equal to.
   - greater than.
   - greater than or equal to.

In addition, there are instructions for branches on overflow and carry resulting from arithmetic operations.

6. Call and Return instructions:

For calling, there are two instructions:

1) "CALLS", where the arguments are pushed onto the stack.

2) "CALLG", where the argument list address is explicitly passed to the routine as an operand.

In our code generation, "CALLS" is used for procedure and function calls. For "CALLG", the following steps are involved:

1) the SP is aligned to a longword boundary by clearing the rightmost two bits after saving the old bits.

2) the registers shown in the register mask are pushed onto the stack.

3) the contents of PC, FP, and AP are pushed onto the stack.

4) a longword containing PSW, the stack alignment bits, and the condition codes is pushed onto the stack.

5) a longword of 0's is pushed onto the stack.

6) the current value of SP is copied to FP.

7) the address of the argument list is copied to AP.

8) the PC is then loaded with the start address of the procedure.

For"CALLS", the following steps are done:

1) the arguments count is pushed onto the stack.

2) the same steps 1-6 of "CALLG" are done.

3) the old value of SP is copied to AP.

4) the address of the procedure is copied to PC.

There is one return instruction to pass control back to the calling environment doing the following:

the information pushed onto the stack in steps 1-5 of "CALLG" is popped in the reverse order. In addition, the return from "CALLS" involves: the arguments count is popped then the actual arguments are popped from the stack by adding to the SP the product of 4 by the arguments count.

7. Character string instructions:

These instructions which manipulate strings and blocks include moving and comparing as follows:

- move character with the two operands of the same size.

- move character with the two operands of different sizes. If the destination operand is longer then it is filled out with the fill character specified in the

instruction.

- compare character with the two operands of the same size.

- compare character with the two operands of different sizes. The excess characters of the longer operand are compared to the fill character specified in the instruction.

Instructions could have 0, 1, 2, 3, 4, 5 or 6 operands. There are no limitations on the types of these operands, they can use any of the addressing modes explained previously.

Chapter IV

OVERVIEW OF CODE GENERATORS

4.1 Introduction

The final stage in the design of a compiler is the code generation which is sometimes called the back end generator. In the analysis-synthesis model of the compiler, the translation of the source code into the intermediate code is called the front end. The back end generates the final code of the target machine from the intermediate code [1]. This chapter surveys the following topics: intermediate languages, forms of object code, memory management, instruction selection, register allocation, and code optimization.

4.2 Intermediate Languages

The intermediate language is a list of simple operations semantically equivalent to the source program [5]. The intermediate languages are represented in different formats including :

- Quadruples
- Triples

- Postfix
- Prefix
- Syntax trees
- Directed acyclic graphs

Following is a brief description of each of the above formats:

Quadruples :

A quadruple expresses a single operation in the form of an operation followed by the two operands and the result. In this form, the operation is applied to the two operands producing a value to be saved as the result [5]. It is also known as three-address code.

Triples :

A triple is a form of a quadruple where the result field is not there. The result is used instead in a later triple [6]. The resulting value is normally kept in one of the working registers. If the result is to be saved, a separate triple is followed with the assignment being the operation.

Postfix :

In this form, the operands come first followed by the operation to be applied to the operands. The postfix

notation is a convenient representation for arithmetic expressions.

Prefix :

In this type of representation, the operation is found followed by the operands. As for postfix, prefix representation is convenient for arithmetic expressions.

Syntax Trees :

A syntax tree is a condensed form of parse trees. The operators and the keywords appear as internal nodes. The operands appear as leaves [1,2]. An operator is applied to its immediate children. The order of evaluation in the syntax trees is bottom-up.

Dags :

A dag denotes a directed acyclic graph. The representation of a dag is similar to that of a syntax tree except that common subexpression are not repeated [1,2]. If the subexpression is already there, an edge from the current node to that subexpression is created.

4.3 Forms of Object Code :

The final object code of a compiler takes a variety of forms:

1. Absolute Machine Language Code.
2. Relocateable Code.
3. Assembly Language Code.

The first choice has the advantage that it is placed in fixed locations in the memory and is immediately executed. On the other hand, the disadvantage is that programs cannot be compiled separately and then linked together for execution. Another disadvantage is the unability to relocate programs since addresses are fixed.

The relocatable code overcomes the disadvantage of the first choice. An additional expense of linking and loading is paid.

The production of assembly language code is the easiest choice. The use of the assembler macros makes this choice a reasonable alternative [1]. Another advantage is that it makes the debugging process easier. The price of this choice over the relocatable code is the cost of the additional step of assembling.

In some special purpose compilers, the target code could be another high level programming language or a special sort of tables or lists.

4.4 Memory Management :

The mapping of the source program variables into actual addresses of the data objects at run-time is normally carried out by the front end generator in cooperation with the back end generator. The type of a given name determines the amount of storage needed for that name. The address given in the front-end for a certain name is relative to the data area at which the name is to be stored. Determination of the data area address is handled in the final code generation according to the class of the name; local, global...etc. This is classified as static memory allocation. The other type of memory allocation is dynamic which is the space allocated at run-time. Dynamic memory is allocated using system defined routines which are part of the run-time support library.

4.5 Instruction Selection :

The selection of the appropriate target language

instructions to implement the various operations can be a large combinatorial task [1,2]. The selection of code is more difficult if the target machine has a rich set of instructions and a large number of addressing modes. The importance of instruction selection comes from the fact that the quality of the produced code depends on its speed and size. Choosing the best sequence of machine instructions for a certain operation requires an extensive knowledge about the context in which the current operation is found. Methods for handling instruction selection such as tree rewriting and pattern matching are found in [1,2,8].

4.6 Register Allocation and Assignment :

Efficient use of registers is an important optimization issue since instructions that have register operands are faster. Register allocation means selecting temporaries and variables that are to be held in registers, while register assignment means specifying which register to use for a selected variable [3]. The amount of research done reflects the importance of register allocation [3,6,7,8,11,15,21].

Register allocation is divided into two steps : local and global. A local register allocator assumes that the program has been partitioned into basic blocks. A basic block is a

sequence of consecutive statements in which the flow of control enters at the beginning and leaves at the end without branching or halting. The variables and the temporaries in a basic block are assigned the set of available registers. Upon exit from a basic block, some variables and temporaries are live. Live variables are those variables that will be used in a subsequent block. Normally, these variables are stored in the memory when leaving a basic block.

Global register allocation keeps the live variables in the registers they are assigned across the basic blocks. The larger the set of blocks used, the better the optimization. If the whole program is used as one unit, the resulting optimization is better. The cost is the complexity of the global register allocator.

When a register is needed for a computation and there is none free, one of the assigned registers is saved in a memory location. The selected register is loaded with the new variable. The process of selecting which register should be saved is called spilling [1].

The problem of global register allocation and spilling is solved by mapping the problem to a graph coloring problem [1]. The nodes of the graph represent variables and

temporaries that need to be assigned to registers. The links between the nodes implies that these nodes need to acquire different registers. The colors denote the set of registers available for assignment. The coloring graph is referred to as the register-interference graph. The graph coloring problem is an NP-complete problem [1]. As a result, many heuristics have been developed to solve this problem in polynomial time. These heuristic algorithms are discussed fully in [3,6,7,8,15,21]. Each of these references describes a different heuristic algorithm to solve the graph coloring problem and hence the register allocation problem. Details of these algorithms are beyond the scope of this work.

4.7 Code Optimization :

The goal of optimization is to produce code that is efficient and optimal. Optimization results in the reduction of the execution time, the code size, and the space requirements. The problem of producing optimal code is undecidable mathematically [1]. Therefore, heuristic techniques are used which do not necessarily generate optimal code. Code optimization can be classified into machine-independent optimization and machine-dependent optimization.

Machine-dependent optimizations utilize the special properties of the target machine. These optimizations include register allocation, which was discussed in the previous section. Another form of machine-dependent optimization is machine idioms. Machine idioms utilize the special machine features such as immediate operands, incrementation, use of indexing, and indirection.

Machine-independent optimizations are mainly based on the mathematical properties of the input and not on the features of the target machine. Some of these optimizations can be done prior to the code generation. Others can be performed either while generating code or after code generation.

Optimizations are said to be local when they are carried out in a basic block. When they are performed with respect to the entire program, they are called global optimizations.

Peephole optimization is a well-known type of optimization [1]. It examines a short sequence of target code (called peephole) and replaces some of these instructions by shorter or faster ones whenever possible. Peephole optimizations include :

- Redundant instruction elimination (e.g. stores and loads).

- Removal of unreachable code.
- Removal of unnecessary jumps.
- Elimination of null operations (e.g. adding zero and multiplying by one).
- Reduction in strength which replaces expensive operations by cheaper ones (e.g. multiplication by additions and exponentiation by multiplication).

Another form of optimization is called folding. In this optimization, some expressions are evaluated at compile-time instead of generating code for them [10]. Expressions evaluated at compile-time have constant operands.

Rearrangement of the order in which operations are carried out is also an optimization method. Here, operations are represented by dags. Selection of an ordering requires that the relationships between the edges are preserved. An optimal ordering is found to be a tree labeling. In a tree labeling, each node of the tree is labeled with an integer denoting the fewest number of registers needed to evaluate the tree. This method labels the tree bottom-up. The output code is generated by traversing the tree in the order of the node labels. Details of this algorithm are found in [1,10].

Loops are optimized by moving invariant operations outside the loop. An operation is said to be invariant in a

loop if it is not affected by changing loop variables.

Common subexpression removal is another form of optimization. These subexpressions are factored out automatically if the basic blocks are represented by the use of dags.

Yet another form of optimization is performed using the algebraic properties of expressions. These properties include the commutative law, the associative law and many others. In logical expressions, short circuiting is used for optimization.

Chapter V

THE FINAL CODE GENERATOR

5.1 Introduction

The final code generator (FCG) is the third module in the DHAD compiler. The compiler is implemented using Pascal. It was originally developed on a Z-80 based computer system; the code generated was Z-80 assembly language. The objective of this research is to rewrite the final code generator to produce assembly language code for VAX machines running under the VMS operating system. The work done can be divided into the following stages:

1. Studying the final code generation module for the Z-80 micro-computer system.
2. Rewriting FCG to produce VAX assembly language code.
3. Using Run-time system support libraries for some functions especially I/O functions and procedures.
4. Changes to other modules.
5. Testing the program against programming examples to make sure that it performs correctly.

5.2 The Code Generation Module

The first task is understanding the structure of the input file which is the intermediate file produced by the second module of the compiler. The intermediate file is presented in (sec 5.3). The next task was to trace some programs to see the operations done to produce Z-80 machine code. This stage took quite a lot of time because tracing, in general, is very difficult especially for programs written by others. In this stage we noticed the following points:

- The Z-80 processor has a word length of 8 bits and a 16-bit address.
- Operations on memory assume that the address is stored in the H-L register pair.
- The operands of integer arithmetic and comparison operations are stored in the H-L pair and the D-E pair and the result of performing the operation is stored in the H-L pair for arithmetic and in the flags for comparison.
- The use of system-defined macros to implement real arithmetic, comparison, string, and some logical operations.
- The use of library routines to perform I/O functions and standard functions.

- The size of integers is two bytes and the size of bytes, words, booleans is one byte. Pointers have the same size as integers while that of reals is four bytes.
- Some operations require one operand to be register A.

5.3 The Intermediate File

The contents of the intermediate file have a couple of formats for the various types of operations. These formats include triples, postfix and prefix.

Triples are used to represent the assignment operations with an opcode followed by the the two operands. The first of which is a variable to be assigned. The second operand is either a constant value, a single variable, or a result of the previous operation available in one of the registers. Preparing an operand is represented by a form of a triple having only one operand.

Arithmetic and comparison operations are represented by the postfix notation. The operands are prepared first and the operation to be applied to the operands is then emitted. The result is used in a later step.

The prefix notation is used to represent the input and output operations.

For procedures and functions, the parameters are first prepared followed by the call to the respective routine. In case of functions, the function value is moved in the next operation.

An operand has the following information:

1. the level, which holds the nesting level. The level of the main program is zero and an increase of one is given to each subsequent nesting.
2. the type, which holds the type of the operand: integer, real,....etc.
3. the class, which indicates whether it is simple, complex, indirect or constant. It also indicates whether it is a global, local, non-local, constant, or variable parameter.
4. the address, which is the operand's location in memory with respect to the frame pointer. For complex types like arrays, for example, the base address of the array is given.
5. the displacement, which gives the position of the operand inside the structure if the type is complex.
6. the length of the operand in bytes.

7. the value of the constant if the class is constant.

5.4 Rewriting the Final Code Generation Module

The focus of this research is to rewrite the final code generator module to produce VAX assembly language code. This module takes as input the intermediate file produced by the syntax and semantics module, then produces the assembly code. Although the work is concentrated towards the code generation module many changes were done in the first two modules: the lexical module and the syntax and semantics module. These changes include:

- changing the integer length from two bytes to four bytes.
- increasing the word length to two bytes instead of one.
- incrementing the pointer length to four bytes, since addresses occupy 32 bits.
- for arrays, the base address is changed to be the lowest address of the array instead of the highest address to make direct use of some of the instructions like move character, compare character, and other character operations.
- correcting some bugs of the first and the second

modules.

In the code generation module, the operations can be classified into:

* integer operations.
* floating point (real) operations.
* byte operations.
* logical operations.
* string operations.
* call, return, and parameter preparation operations.
* conditional and unconditional branch operations.
* input and output operations.
* address preparation.
* register allocation.

For each of these operations we will discuss in full detail their types, form of input, machine code generated and registers used to hold their operands. The first byte read from the intermediate file is the opcode whose value is used to transfer control to a specific routine.

5.4.1 Integer Operations:

In the integer operations module, the opcode classifies the kind of operation to be performed. For each kind of

operation, the required parameters are read from the intermediate file. The various operations that can be performed on integers are:

- **Integer Preparation :**

  This operation requires only one operand. The operand is prepared by putting it in a register to be used by the next operation. The code to be generated is:

  MOVL address(FP),Rn

  Determination of Rn will be discussed in register allocation.

- **Assignment :**

  This operation requires two operands. if the second operand is a constant then a direct move puts this constant value into the memory location indicated by the first operand. The code generated is in the following form:

  MOVL #constant,address1(FP) if class is simple

  MOVL #constant,(Rn) if class is complex.

  Rn contains the address of the specific element of the complex structure.

If the second operand is not a constant value then we prepare that operand in a register. Then a move of the

prepared operand to the designated address given in the first operand is performed. The code generated is:

```
MOVL  address2(FP),Rn
MOVL  Rn,address1(FP)  if class is simple.
```

If class is complex then address(FP) is replaced by (Rm).

- **Convert :**

This operation converts from an integer to a real. The operand to be converted is already prepared and ready in a register. The code generated is:

```
CVTLF   Rn,Rn
```

- **Negate :**

This operation negates an operand which is already available in a register. The generated code is:

```
MNEGL   Rn,Rn
```

- **Arithmetic :**

This operation comes without operands since the operands must have already been prepared in registers, Rn contains the first operand and Rm contains the second operand, and the result is stored in Rn. The code produced is one of the following:

```
ADDL2     Rm,Rn    or
SUBL2     Rm,Rn    or
MULL2     Rm,Rn    or
```

```
DIVL2      Rm,Rn
```

- **Comparison :**

This operation is the same as the arithmetic but results are not stored; instead, flags are set accordingly. The produced code is:

```
CMPL     Rn,Rm
```

The use of the results (flags) will be discussed in conditional branch and logical assignment.

- **Complex Preparation and Address Preparation :**

This operation will be discussed in detail in separate sections (secs 5.4.9 and 5.4.10) since it is common for all types of data.

5.4.2 Real Operations :

In the real operations module, like in integer operations, the opcode determines the parameters to be read from the intermediate file. The various operations that can be done on reals are:

- **Real Preparation :**

This operation is the same as integer preparation since in VAX assembly language, registers can be used for integers or reals. The only difference is the code generated which

is:

```
MOVF      address(FP),Rn
```

- **Assignment :**

This operation is also similar to the integer assignment but the code produced is different. In the case that the second operand is a constant value, It generates the following:

```
MOVF  #constant,address1(FP) if class is simple, and
MOVF  #constant,(Rn) if class is complex.
```

Rn contains the address of the specific element of the complex structure.

If the second operand is not a constant value then we produce:

```
MOVF  address2(FP),Rn
MOVF  Rn,address1(FP)  if class is simple.
```

If class is complex then address(FP) is replaced by (Rm).

- **Negate :**

This operation negates an operand which is already available in a register. The generated code is:

```
MNEGF   Rn,Rn
```

- **Arithmetic :**

This operation comes without operands since the operands should already have been prepared in registers, Rn contains the first operand and Rm contains the second operand, and the result is be stored in Rn. The code produced is one of the following:

```
ADDF2     Rm,Rn     or
SUBF2     Rm,Rn     or
MULF2     Rm,Rn     or
DIVF2     Rm,Rn
```

- **Comparison :**

This operation is the same as the arithmetic but results are not stored; instead, flags are set accordingly. The produced code is as follows:

```
CMPF      Rn,Rm
```

- **Complex Preparation and Address Preparation :**

As mentioned earlier, this operation will be discussed in detail in (secs 5.4.9 and 5.4.10) since it is common for all types of data.

5.4.3 Byte and Word Operations :

In the byte operations module, like in integer operations, the opcode determines the parameters to be read from the intermediate file. The byte and word operations are exactly the same except that a word is two bytes. The various operations that can be done on bytes and words are:

- **Preparation :**

This operation is the same as integer preparation. The only difference is that we zero out the high_order bytes of the register used. It produces the following code:

```
MOVZBL   address(FP),Rn   if type is byte,
MOVZWL   address(FP),Rn   if type is word
```

- **Assignment :**

This operation is also similar to the integer assignment but the code produced is different. The high_order bytes of the register used are cleared only if the operation involves registers. In the case that the second operand is a constant value, It generates the following code for bytes:

```
MOVB  #constant,address1(FP) if class is simple
MOVB  #constant,(Rn) if class is complex.
```

Rn contains the address of the specific element of the

complex structure.

For the case of words, the following code is produced:
MOVW #constant,address1(FP) if class is simple
MOVW #constant,(Rn) if class is complex.
Rn contains the address of the specific element of the complex structure.

If the second operand is not a constant value then for bytes, we produce:
MOVZBL address2(FP),Rn
MOVB Rn,address1(FP) if class is simple.
If class is complex then address(FP) is replaced by (Rm).

In the case of words, the same code as the one produced for bytes is generated with B (byte) changed to W (word). The code looks as follows:
MOVZWL address2(FP),Rn
MOVW Rn,address1(FP) if class is simple.
If class is complex then address(FP) is replaced by (Rm).

- **Negate :**

This operation negates an operand which is already available in a register. The generated code is:

```
MNEGB   Rn,Rn      for bytes and
MNEGW   Rn,Rn      for words
```

- **Complement :**

This operation is done by taking the ones complement of the byte or the word already prepared in a register. The code generated is the following:

```
MCOMB       Rn,Rn   in case of bytes
MCOMW       Rn,Rn   in case of words.
```

- **Arithmetic :**

This operation comes without operands since the operands are assumed to have been prepared in registers; Rn contains the first operand and Rm contains the second operand, and the result is stored in Rn. The code produced is one of the following in case of bytes:

```
ADDB2     Rm,Rn    or
SUBB2     Rm,Rn    or
MULB2     Rm,Rn    or
DIVB2     Rm,Rn
```

In the case of words the code is:

```
ADDW2     Rm,Rn     or
SUBW2     Rm,Rn     or
MULW2     Rm,Rn     or
DIVW2     Rm,Rn
```

- **Logical AND :**

This operation cannot be done in one step since VAX does not provide a direct instruction to AND two operands but instead it has an instruction to AND the ones complement of the second operand with the first operand. So we first complement the second operand (Rm) and then we AND it with the first (Rn). The result is stored in the first operand which is Rn in this case. The code for bytes is:

```
MCOMB     Rm,Rm
BICB2     Rm,Rn
```

The code for words is:

```
MCOMW     Rm,Rm
BICW2     Rm,Rn
```

- **Logical OR :**

This operation finds the logical OR of two operands which are already prepared in registers Rn and Rm. The code produced for bytes is:

```
BISB2     Rm,Rn
```

In the case of words the code is:

```
BISW2    Rm,Rn
```

- **Exclusive-OR :**

This operation finds the logical exclusive OR of two operands which are already prepared in registers Rn and Rm. The code produced for bytes is:

```
XORB2    Rm,Rn
```

In the case of words the code is:

```
XORW2    Rm,Rn
```

- **Comparison :**

This operation is the same as the arithmetic operations but results are not stored; instead, flags are set accordingly. The produced code is:

```
CMPB     Rn,Rm    for bytes
CMPW     Rn,Rm    for words
```

- **Complex Preparation and Address Preparation :**

This operation will be discussed in detail in (secs 5.4.9 and 5.4.10) since it is common for all types of data.

5.4.4 Boolean (Logical) operations :

In the Boolean operations module, like in integer operations, the opcode determines the parameters read from

the intermediate file. The Boolean operations are exactly the same as byte operations since a true value is stored as one in a byte and a false value is stored as zero in a byte also. However more operations are available for Booleans in logical expression, in selection and looping constructs. Short circuiting is used in evaluating logical expressions. This reduces the run time. Arithmetic operations are not allowed for Boolean types. The operations done for Booleans are:

- **Preparation :**

This operation is the same as byte preparation. It produces the following code:

MOVZBL address(FP),Rn

- **Assignment :**

This operation is also similar to the byte assignment and the code produced is the same. In the case of the second operand being a constant value, the following code is generated:

MOVB #constant,address1(FP) if class is simple

MOVB #constant,(Rn) if class is complex.

Rn contains the address of the specific element of the complex structure.

If the second operand is not a constant value then we produce:

```
MOVZBL  address2(FP),Rn
MOVB  Rn,address1(FP)  if class is simple.
```

If class is complex then address(FP) is replaced by (Rm).

- **Testing :**

This operation is done by testing the operand to determine whether it is a one or zero. The code generated is:

```
TSTB      operand
```

The operand here could be a register or a memory location.

- **Anding :**

This operation produces a conditional branch instruction according to the flags previously set by the test instruction. This branch determines at run time whether to continue testing the other operand or to quit if the first operand is zero. The code produced by this operation is:

```
BEQL      label
```

- **Oring :**

This operation produces a conditional branch instruction according to the flags previously set by the test instruction. This branch determines at run time whether to continue testing the other operand or to quit if the

first operand is one. The code produced by this operation is:

```
BNEQ     label
```

- **Complement :**

This operation is done by changing one into zero and changing a zero to one. If we are complementing an expression then we exchange the ANDs with ORs and the ORs with ANDs. Also the branch conditions are reversed "Demorgan's Law".

- **Comparison :**

This operation comes without operands since the operands must have been prepared in registers; Rn contains the first operand and Rm contains the second operand, and the result of this operation is setting the flags. The code produced is the following:

```
CMPB     Rn,Rm
```

- **Complex Preparation and Address Preparation :**

This operation will be discussed in detail in (secs 5.4.9 and 5.4.10) since it is common for all types of data.

5.4.5 String and Block operations :

String operations require special handling. The actual operand cannot be put in registers but instead its address is moved into a register. A string occupies one more byte than its declared length and the extra byte is used to store the actual length of the string since strings are of variable length. The length is stored in the first byte of the string. The other blocks like arrays and records can be moved using string assignment operation. The operations for strings are:

- **Preparation :**

This operation pushes the address of the string being prepared onto the stack. The address of the start of the string is calculated by subtracting the length from the address given as part of the operand read from the intermediate file. The following code is generated by this instruction:

```
MOVL   FP,Rn     FP is the frame pointer
ADDL2  #address-length,Rn
PUSHL  Rn
```

- **Block Assignment :**

This operation moves a block of data from the address given by the second operand to the address given by the

first operand. In the case of the second operand being a constant value string the following code is generated:

```
        BRB     Sm
Sn:     .ASCII  \constant value\
Sm:     MOVL    #Sn,Rm
        MOVB    #length,(Rn)
        MOVC3   #length,(Rm),(Rn)
```

The address of the first operand is prepared exactly like the string preparation explained in the previous section.

If the second operand is not a constant value string and the type is string then we produce:

```
MOVB    #length,(Rn)
MOVC3   #length,(Rm),(Rn)
```

The addresses of the first and the second operands are prepared as in the string preparation explained in the previous section. They are stored in registers Rn and Rm respectively.

If the type is not string then the code generated is:

```
MOVC3   #length,(Rm),(Rn)
```

- **Comparison :**

For this operation, the addresses of the operands must already have been pushed onto the stack. The address of the second operand is first pushed onto the stack then the address of the first operand. The effect of this operation is setting the condition flags. The code produced is the following:

```
CMPC5   #length1,-4(SP),#A/ /,#length2,(SP)
```

- **Complex Preparation and Address Preparation :**

This operation will be discussed in details in (secs 5.4.9 and 5.4.10) since it is common for all types of data.

5.4.6 Call and Return operations :

In the call and return operations, invocation and creation of functions and procedures are discussed. These operations start with preparing the parameters, if any, by pushing their values or their addresses on the stack. After the parameters are ready a call to the routine, function, or procedure is initiated along with the number of parameters. In addition,in this section, the creation of a call frame, the return from routines, quitting any looping construct in the program and stopping the execution of a program are discussed. The types of

operations done are:

- **Quit :**

This operation jumps unconditionally to the end of the loop in which the quit occurred. This operation requires one parameter, the label for the jump, read from the intermediate file. Normally this is preceded by a test for a certain condition to go out of the loop. It generates the following code:

```
JMP   label
```

- **Stop :**

The stop operation terminates the program and it returns a value of one in register R0 indicating successful completion of the program (VAX convention). This operation occurs only once in any program. If a stop is encountered in the middle of a program, it is converted to an unconditional jump to the end of the program. The produced code is:

```
  $EXIT_S
  .END  main program name
```

- **Return :**

This operation causes a return from the routine to the calling environment. Upon return the saved registers (SP, FP, PC, AP) are restored and the parameters previously

pushed on the stack are cleared. If the routine is a function, then the function value is returned in the stack. The following code is generated:

```
RET
```

- **Parameter Preparation :**

This operation prepares a parameter for the function or procedure call. If the class is constant (passing by value), then the parameter is prepared according to its type, and the prepared parameter is pushed onto the stack. If the class is variable (passing by address), then the address of the parameter is prepared and pushed onto the stack. The code produced is first preparing the value or the address according to its type in a register, then:

```
PUSHL    Rn
```

- **Function Prepare :**

This operation allocates space for the name of the function itself as a variable since function names have values. This space is allocated in the stack before the parameters. The following is produced:

```
SUBL2    #length,SP
```

- **Procedure Call :**

This operation executes a call to the respective procedure along with the number of parameters. If the procedure is

' جديد ' then this is a call to allocate dynamic space for pointers. The address of the memory location in which the allocated space address is stored is pushed onto the stack. Before a call is initiated the length of the needed memory is also pushed onto the stack. Then the call is started returning the address of the allocated space in register R0. This address is moved into the pointer address previously pushed onto the stack and the stack is cleared from this entry. The code generated looks like:

```
PUSHL    #length
CALLS    #1,PAS$NEW2
MOVL     (SP)+,Rn
MOVL     R0,(Rn)
```

If the procedure is ' اطلق ' then this is a call to deallocate a previously allocated dynamic space. The address of this space is pushed onto the stack in a previous step, and the call is initiated with the following code:

```
CALLS    #1,PAS$DISPOSE2
```

In the case of user procedures, a static pointer is prepared and stored in a fixed location from the current

frame pointer (FP). The preparation of the static pointer is carried out in the following way:

If the level of the current procedure is equal to the level of the called procedure, the current static pointer is copied to the static pointer of the called procedure producing this code:

```
PUSHL   24(FP)
```

If the level of the current procedure is less than that of the called procedure, the value of FP is copied to the static pointer of the called procedure generating:

```
PUSHL   FP
```

In case the level of the current procedure is greater than that of the procedure to be called, a move to the correct static pointer is carried out. The move is backwards through the chain of the static pointers. The number of levels to be crossed is equal to the difference between the current level and the called procedure level. When the designated level is reached, its static pointer is copied to the static pointer of the called procedure. It produces the following code:

```
MOVL    24(FP),Rn
MOVL    24(Rn),Rn  repeated as many times as the difference
PUSHL   Rn
```

After the static pointer is prepared and saved, the call is initiated along with the number of parameters already pushed onto the stack producing:

```
CALLS   par_num,procedure
```

- **Function Call :**

This operation is similar to the procedure call except that there is a value returned in the name of the function. This value is in the top of the stack to be moved to register Rn if the type is simple; otherwise, it is moved from the stack in a later step. The following piece of code is generated:

```
CALLS   par_num,function
```

If the function type is integer, for example, then it produces:

```
MOVL    (SP)+,Rn
```

Other types are handled similarly.

- **Procedure-function Definition :**

This operation defines a procedure or a function and it allocates local space to be used by that procedure or function. The procedure label and the number of bytes to be allocated are read from the intermediate file. The following code is generated:

```
.ENTRY   procedure label, M<>
```

```
SUBL2     number of bytes,SP
```

5.4.7 Conditional and Unconditional Branch Operations :

In this section, the operations of defining labels, unconditional branch instructions, and conditional branch instructions are discussed. In all these operations, the label number is read from the intermediate file. The actual label is defined by prefixing the label number with the letter 'C'. These operations include:

- **Unconditional Jump :**

  This operation generates an unconditional jump to the label read from the intermediate file. The following code is produced:

```
JMP       C_label
```

- **True Jump :**

  This operation uses the branch number previously set when the comparison was encountered. The branch number is an integer between one and six. It identifies the condition needed for the jump to be performed. Here the branch occures if the condition is true. The code produced is one of the following:

```
BEQL      C_label
```

```
BNEQ     C_label
BGEQ     C_label
BLEQ     C_label
BLSS     C_label
BGTR     C_label
```

- **False Jump :**

  This operation generates a conditional branch instruction according to the branch number. The branch is carried out if the condition is false. The code produced is one of the following:

```
BEQL     C_label
BNEQ     C_label
BGEQ     C_label
BLEQ     C_label
BLSS     C_label
BGTR     C_label
```

- **Unconditional Label :**

  This operation defines a label for an unconditional branch instruction with the following code:

```
C_label:
```

- **True Label :**

  This operation defines a label and produces the following code:

C_label:

After defining the label then we check the or_stack to see if there are any labels used in a previous OR instruction not yet defined. If any labels are found then they are equated to the current label. The label for the or_stack is defined by prefixing 'B' to the label number. The code for these looks like:

B_label = C_label

- **False Label :**

This operation defines a label and generates the following code:

C_label:

After defining the label then we check the and_stack to see if there are any labels used in a previous AND instruction not yet defined. If any labels are found then they are equated to the current label. The label for the and_stack is defined by prefixing 'B' to the label number. The code for these looks like:

B_label = C_label

5.4.8 Input and Output :

In this section input and output from both the screen and files are discussed. The opening of files and their preparation for reading and writing are also explained. The operations done here include the following types:

- Console Write
- Console Read
- Reset_Rewrite
- File Read
- File Write

These types of operations are discussed in full detail as follows:

Console Write :

This sub_module prepares parameters to be printed on the screen by first preparing the operand in a register then pushing that operand onto the stack if the operand type is simple. If the operand type is complex then it is directly pushed onto the stack. If the operand is an expression, the expression is first evaluated and the result is pushed onto the stack. The formatting information is read from the intermediate file and pushed also onto the stack. Next, the address of the output buffer is pushed onto the

stack and the call to the respective routine is initiated. The code generated for printing integers is:

```
PUSHL   Rn
PUSHL   width
PUSHAB  PAS$FV_OUTPUT
CALLS   #3,PAS$WRITE_INTEGER
```

The code generated for printing bytes,words and logical is:

```
PUSHL   Rn
PUSHL   width
PUSHAB  PAS$FV_OUTPUT
CALLS   #3,PAS$WRITE_UNSIGNED
```

The code generated for printing reals is:

```
PUSHL   Rn
PUSHL   width
PUSHL   fraction
PUSHAB  PAS$FV_OUTPUT
CALLS   #4,PAS$WRITE_REALF_F
```

The code generated for printing strings is:

```
PUSHL   string_address
```

```
PUSHL  width
PUSHL  #0
PUSHL  length of string
PUSHAB PAS$FV_OUTPUT
CALLS  #5,PAS$WRITE_STRING
```

console read :

In this sub_module, the reading process can be divided into two parts according to the type of parameter to be read. The first part is for simple types like integers, reals, words, bytes and Boolean. The second part is for strings. For the case of simple types, the address of the input buffer is first pushed onto the stack then the call to the reading routine is initiated, which performs the actual read returning the result in register R0. This convention of returning the value read in register R0 is used by the run time support library which is used to perform the reading operation. After the value is read into R0, the address of the actual operand is prepared according to its type. At this time the contents of register R0 are moved to the address prepared.

The code generated for integers is:

```
PUSHAB PAS$FV_INPUT
CALLS  #1,PAS$READ_INTEGER
```

The address of the operand is prepared in register Rn as explained previously.

```
MOVL    R0,(Rn)
```

The code generated for reals is:

```
PUSHAB PAS$FV_INPUT
CALLS  #1,PAS$READ_REALF_F
```

The address of the operand is prepared in register Rn as explained previously.

```
MOVF    R0,(Rn)
```

The code generated for bytes and Booleans is:

```
PUSHAB PAS$FV_INPUT
CALLS  #1,PAS$READ_UNSIGNED
```

The address of the operand is prepared in register Rn as explained previously.

```
MOVB    R0,(Rn)
```

The code generated for words is:

```
PUSHAB PAS$FV_INPUT
CALLS  #1,PAS$READ_UNSIGNED
```

The address of the operand is prepared in register Rn as explained previously.

```
MOVW    R0,(Rn)
```

For the case of strings, the picture is a bit different. The first thing done is pushing the maximum length of the string onto the stack. The address of the string is prepared and then pushed onto the stack followed by pushing the address of the input buffer. At this time the call to the reading routine is initiated which reads the actual value and stores it. The code produced for this operation is:

```
PUSHL   max_length
PUSHL   string_address
PUSHAB  PAS$FV_INPUT
CALLS   #3,PAS$READ_VARYING
```

Reset-Rewrite :

In this sub_module the issue of preparing a file for read or write is handled. Since we do not have an operation to open a file, opening files is implicit with a reset or a rewrite operation. Files opened by a reset operation are assumed to be old and files opened with a rewrite operation are assumed to be new. This operation requires two operands, the first of which defines the file address and size whereas the second operand defines the name of

the file. The first thing done is pushing the file's status (old or new) onto the stack. This step is followed by preparing the name of the file. If the name is constant, it is defined and its address is pushed onto the stack. If the name is variable, its address is prepared and pushed onto the stack. The length of the file name is then pushed onto the stack. After that, the file address is prepared and the file variable is defined with the required parameters. The last thing pushed onto the stack is the file variable address after which the open routine is invoked. All the previous steps are common to both reset and rewrite. Now, if the operation is reset then the file variable address is pushed onto the stack followed by a call to the library routine reset. If the operation is a rewrite, the address of the file variable is pushed then the library routine rewrite is called. The code produced by this operation is:

```
PUSHL   status
PUSHL   address of file name
PUSHL   length of file name
MOVL    Rn,R0
MOVAB   28(R0),(R0)+
CLRL    (R0)+
MOVAB   8(R0),(R0)+
```

```
CLRL    (R0)+
CLRL    (R0)+
MOVL    #2,(R0)+
MOVL    record length,(R0)  for non-text files
PUSHL   Rn
CALLS   #5,PAS$OPEN2
PUSHL   Rn
```

If the operation is reset then it produces:

```
CALLS   #1,PAS$RESET2
```

If the operation is rewrite then it produces:

```
CALLS   #1,PAS$REWRITE2
```

File Write :

This operation prepares the parameter to be written to the file in register Rn and moves it to the file buffer. The address of the file variable is prepared in register Rm and is pushed onto the stack followed by initiation of the call to a routine to copy the contents of the buffer into the actual file. The code produced if the file type is simple is:

```
MOVL   Rn,28(Rm)
PUSHL  Rm
CALLS  #1,PAS$PUT
```

The code produced if the file type is complex is:

```
MOVC3   record length,(Rn),25(Rn)
PUSHL   Rm
CALLS   #1,PAS$PUT
```

File Read :

This operation first pushes the address of the file variable, which is prepared in register Rm, onto the stack and it initiates the call to a routine that copies the record needed from the actual file to the file buffer. Then it moves the file buffer into the address of the parameter after preparing it in register Rn. The code produced if the file type is simple is:

```
PUSHL   Rm
CALLS   #1,PAS$GET
MOVL    28(Rm),Rn
```

The code produced if the file type is complex is:

```
PUSHL   Rm
CALLS   #1,PAS$GET
MOVC3   record length,25(Rm),(Rn)
```

5.4.9 Address Preparation :

the address calculated for any memory reference is actually the displacement from a given base address stored in a register. Deciding and preparing the register to be used as the base depends on the class of that memory reference. The class is either global, local, non_local, constant_parameter, variable_parameter. Each of these classes is associated with a certain register to hold the address of base suitable to the memory reference being used.

Global references use register R10 as their base address. Register R10 is loaded with the address of the global data area. Actual addresses are calculated relative to R10.

Register FP (Frame Pointer) is used to reference variables which are local. When any procedure or function is invoked, the old value of FP and the values of the needed registers are pushed onto the stack, and FP is loaded with the address of the top of the stack creating a new frame which is called the call frame. This value of FP is used as the base for local references.

For the non_local references, a static pointer is

maintained in a fixed location relative the the current FP. Whenever a call to a procedure or a function is encountered, the static pointer is prepared and saved in its location. Preparing the static pointer was discussed in the procedure call.

For the usage of the non_local references, the current static pointer is used if the reference level is greater than or equal to the procedure level. If the reference level is less than the procedure level, the current static pointer is used to chain through the static pointers a number of levels equal to the difference. Reaching the required level, the static pointer of this level is used as a base address for the non_local reference at hand. It generates the following code:

```
MOVL    24(FP),Rn
MOVL    24(Rn),Rn
```

The number of times the second statement is repeated is one less than the difference between the procedure level and the reference level.

For constant_parameters, the register AP (Argument Pointer) with the given displacement to access the actual value needed is used.

For variable_parameters, the register AP is also used with the difference that the displacement from AP accesses the address of the memory reference. This address in turn is used to access the actual value.

For the case of indirect addresses, the indirect addressing mode is used with the appropriate address.

5.4.10 Complex Address preparation :

Preparing an address for an element of a complex structure like an array requires special handling. The address calculation for one-dimensional arrays is done as follows: Assume that W is the size of the array element (depends on the type), BASE is the starting address of the array, LOW is the lower bound of the array, and INDEX is the index of the element whose address is to be calculated. The element address equals:

BASE + (INDEX - LOW) * W or

BASE + INDEX * W - LOW * W

For the second equation, BASE - LOW * W is a constant value that can be calculated at compile time. This value is given in the intermediate file as the address of the array. The operations done in the code generator are to

multiply the index by the size of the element and add the result to the address read from the intermediate file. The generated code is:

```
MOVL   INDEX,Rn
ADDL2      Rn,Rn
ADDL2      Rn,Rn
MOVL       address,Rm
ADDL2      Rn,Rm
```

The size of the element in the previous code is four and therefore, the multiplication is changed to addition.

The address calculation for two-dimensional arrays is carried out in the following way:
Assume that W is the size of the array element (depends on the type), BASE is the starting address of the array, LOW1 and LOW2 are the lower bounds of the array,UP is the upper bound of the column index, INDEX1 and INDEX2 represent the row and the column indices of the array element, and LW is the size of the row in bytes. The element address equals:

BASE + (INDEX1 - LOW1) * LW + (INDEX2 - LOW2) * W or
BASE - LOW1 * LW - LOW2 * W + INDEX1 * LW + INDEX2 * W
where LW = W * (UP - LOW2)

In the second equation, the value of BASE-LOW1*LW-LOW2*W can be calculated at compile time since it is

constant. This value is calculated and is ready in the intermediate file as the address of the array. The sequence of operations are:

First, multiply INDEX1 by the row length LW. Second, multiply INDEX2 by the element length. The results of the two operations are added and finally the address read from the intermediate file is added to the result of the addition. The produced code is:

```
MOVL    INDEX1,Rn
MOVL    LW,Rm
MULL2   Rm,Rn
MOVL    INDEX2,Rm
MULL2   W,Rm
ADDL2   Rm,Rn
MOVL    address,Rm
ADDL2   Rm,Rn
```

In the case of three-dimensional arrays, the address calculation is performed in a similar way.

The address calculation is carried out leaving the result in register R11. This address is kept in R11 to be used in a later step.

5.4.11 Register Allocation :

In our code generator, a simple register allocator is used. Selecting an optimized register allocator is beyond the scope of this thesis and is suggested as future work. The registers are used in the following way:

R0 is used to hold a return value (which is one for successful termination of the main program according to the VAX convention).
R1..R5 are not used since their contents are altered by some instructions.
R6 and/or R7 hold the operands of the arithmetic and the compare instructions.
R8 holds the address of the file variable.
R9 is used to hold the result of evaluating a logical expression.
R10 holds the base address of the global data area.
R11 is used to hold the address of a complex memory reference after preparing it.

The registers are allocated whenever the need arises. If a register is required for a computation, the register availability list is checked. If the requested register is free, it is allocated and marked as not free. If the needed register is found to be allocated, the current

value of the register is saved on the stack. The register is then loaded with the new value. When the new operation is finished, the register value is restored from the stack.

5.4.12 Testing and Validation :

The code generator was implemented using Pascal. It was tested using many examples. These examples included testing of all kinds of operations and constructs. The output of these examples, which is VAX assembly code, was assembled, linked, and then executed. The results had proven the correctness of the generated code. These examples and their corresponding assembly output are listed in the appendix.

Chapter VI

CONCLUSION

The final code generator was rewritten to generate VAX assembly code. The same algorithm for code generation was used with some modifications. The module produced code for all types of operations on the various data structures which include byte, word, integer, real, logical, strings, records, arrays, pointers, and files. Code for operations of branching, I/O, and functions and procedures has been generated. The Pascal run-time support library routines were used for standard functions and procedures and for input and output operations. Some minor changes to the other modules of the compiler, lexical and syntax and semantics analyzers, were needed. The final code generator was implemented using Pascal on the VAX/VMS system. The size of the executable module of the final code generator was 77 blocks which is about 40 Kbytes.

Future Work

The following points are suggested for future work:

- Improving the generated code using various code

optimization techniques.

- Testing the compiler with more programs.
- Rewriting the compiler to produce relocatable code.
- Porting the compiler to other computer systems.
- Improving the register allocation algorithm.

REFERENCES

[1] A. V. Aho, R. Sethi and J. D. Ullman, ***Compilers: Principles, Techniques and Tools***, Addison-Wesley, 1987.

[2] A. V. Aho and J. D. Ullman, ***Principles Of Compiler Design***, Addison-Wesley, 1979.

[3] P. Anklam, D. Cutler, R. Heinen and M. D. Maclaren, ***Engineering A Compiler: VAX-11 Code Generation And Optimization***, Digital, 1982.

[4] S. Baase, ***VAX-11 Assembly Language Programming***, Prentice-Hall, 1983.

[5] W. A. Barrett and J. D. Couch, ***Compiler Construction: Theory and Practice***, Science Research Associates, 1979.

[6] D. Bernstein and others, "Spill Code Minimization Techniques for Optimizing Compilers", Proceedings of the SIGPLAN'89 conference on Programming Language Design and Implementation, SIGPLAN Notices, Vol. 24, No. 1, pp.258-263, June 1989.

[7] P. Briggs, K. D. Cooper, K. Kennedy and L. Torczon, "Coloring Heuristics for Register Allocation", Proceedings of the SIGPLAN'89 conference on

Programming Language Design and Implementation, SIGPLAN Notices, Vol. 24, No. 1, pp.275-284, June 1989.

[8] H. Emmelmann, F. W. Schroer and R. Landwehr, "BEG - A Generator for Efficient Back Ends", Proceedings of the SIGPLAN'89 conference on Programming Language Design and Implementation, SIGPLAN Notices, Vol. 24, No. 1, pp.227-237, June 1989.

[9] C. W. Fraser, "A Language for Writing Code Generators", Proceedings of the SIGPLAN'89 conference on Programming Language Design and Implementation, SIGPLAN Notices, Vol. 24, No. 1, pp.238-245, June 1989.

[10] D. Gries, ***Compiler Construction For Digital Computers***, John-Wiley & Sons, 1971.

[11] R. Gupta, M. L. Soffa and T. Steele, "Register Allocation Via Clique Separators", Proceedings of the SIGPLAN'89 conference on Programming Language Design and Implementation, SIGPLAN Notices, Vol. 24, No. 1, pp.264-274, June 1989.

[12] C. A. Kapps and R. L. Stafford, ***VAX Assembly Language And Architecture***, PWS, 1985.

[13] M. G. Khayat, ***The Arabic Programming Language DHAD And Its Compiler***, CCSE Tech Report, ICS-002, KFUPM, 1988.

[14] K. A. Lemone and M. E. Kaliski, ***Assembly Language Programming For The VAX-11***, Little Brown and Company, 1987.

[15] B. W. Leverett, ***Register Allocation In Optimizing Compilers***, UMI research press, 1983.

[16] P. M. Lewis II, D. J. Rosen Krantz and R. E. Stearns, ***Compiler Design Theory***, Addison-Wesley, 1976.

[17] J. F. Peters, ***The Art Of Assembly Language Programming VAX-11***, Reston, 1985.

[18] G. M. Schneider, R. Davis and T. Mertz, ***Computer Organization And Assembly Language Programming For The VAX***, John-Wiley & Sons, 1987.

[19] R. W. Sebesta, ***VAX-11: Structured Assembly Language Programming***, The Benjamin/Cummings, 1984.

[20] E. F. Sowell, ***Programming In Assembly Language VAX-11***, Addison-Wesley, 1987.

[21] P. A. Steenkiste and J. L. Hennessy, "A Simple

Interprocedural Register Allocation Algorithm And Its Effectiveness For LISP", ACM Transactions On Programming Languages And Systems, Vol. 11, No. 1, pp. 1-32, Jan 1989.

[22] W. M. Waite and G. Goos, ***Compiler Construction***, Spriner-Verlag, 1984.

APPENDIX

```
برنـــامج !!!
<* متغيرات : صحيح : س،ز  *>
<#
        س:= -١٠
        اطبع(س:٥)
        ز:= ٥
        س:= س/ز
        س:= س+ز
        س:= س*س
        ز:= ز-س
        س:=((س-٩)/(ز+٥٤))*-٥
        اطبع('ببببب')
        اقرا (س)
        اطبع (س:١٠،ز:١٠)
        اطبع (ز:١٠)
#>
```

```
        .ENTRY  I.99, M< >
        SUBL2   #8,SP
        MOVL    FP,R10
        MOVL    #10,R6
        MNEGW   R6,R6
        MOVL    R6,0(R10)
        MOVL    0(R10),R6
        PUSHL   #5
        PUSHL   R6
        PUSHAB  G PASSFV  OUTPUT
        CALLS   #3,G PASSWRITE  INTEGER
        MOVL    #5,-4(R10)
        MOVL    0(R10),R6
        MOVL    -4(R10),R7
        DIVL2   R7,R6
        MOVL    R6,0(R10)
        MOVL    0(R10),R6
        MOVL    -4(R10),R7
        ADDL2   R7,R6
        MOVL    R6,0(R10)
        MOVL    0(R10),R6
        MOVL    0(R10),R7
        MULL2   R7,R6
        MOVL    R6,0(R10)
        MOVL    -4(R10),R6
        MOVL    0(R10),R7
        SUBL2   R7,R6
        MOVL    R6,-4(R10)
        MOVL    0(R10),R6
        MOVL    #9,R7
        SUBL2   R7,R6
        MOVL    -4(R10),R7
        PUSHL   R6
        MOVL    #54,R6
        ADDL2   R7,R6
        PUSHL   R6
        MOVL    R7,R6
        MOVL    (SP)+,R7
        MOVL    (SP)+,R6
        DIVL2   R7,R6
        MOVL    #5,R7
        MNEGW   R7,R7
        MULL2   R7,R6
        MOVL    R6,0(R10)
        PUSHL   #4
        PUSHL   #0
        BRB     S1
S0:     .ASCII  \IIII\
```

```
S1:
        PUSHAB  S0
        PUSHL   #4
        PUSHAB  G PASSFV OUTPUT
        CALLS   #5,G PASSWRITE STRING
        PUSHAB  G PASSFV INPUT
        CALLS   #1,G PASSREAD INTEGER
        MOVL    R10,R6
        ADDL2   #0,R6
        MOVL    R0,(R6)
        MOVL    0(R10),R6
        PUSHL   #10
        PUSHL   R6
        PUSHAB  G PASSFV OUTPUT
        CALLS   #3,G PASSWRITE INTEGER
        MOVL    -4(R10),R6
        PUSHL   #10
        PUSHL   R6
        PUSHAB  G PASSFV OUTPUT
        CALLS   #3,G PASSWRITE INTEGER
        MOVL    -4(R10),R6
        PUSHL   #10
        PUSHL   R6
        PUSHAB  G PASSFV OUTPUT
        CALLS   #3,G PASSWRITE INTEGER
C999:
        $EXIT S
        .END    I.99
```

```
برنامج !!!
<* متغيرات : حقيقي: س،ز  *>
<#
        س:= ٠,١٠
        ز:= ٠,٥
        س:= س/ز
        س:= س+ز
        س:= س*س
        ز:= ز-س
        س:=((س-٠,٩)/(ز+٠,٥٤))*-٠,٥
        اطبع('ببيبب')
        اقرا (س)
        اطبع (س:١٠:٥)
        اطبع (ز:١٠:٥)
      #>
```

```
        .ENTRY  L99, M< >
        SUBL2   #8,SP
        MOVL    FP,R10
        MOVF    #10.0,0(R10)
        MOVF    #5.0,-4(R10)
        MOVF    0(R10),R6
        MOVF    -4(R10),R7
        DIVF2   R7,R6
        MOVF    R6,0(R10)
        MOVF    0(R10),R6
        MOVF    -4(R10),R7
        ADDF2   R7,R6
        MOVF    R6,0(R10)
        MOVF    0(R10),R6
        MOVF    0(R10),R7
        MULF2   R7,R6
        MOVF    R6,0(R10)
        MOVF    -4(R10),R6
        MOVF    0(R10),R7
        SUBF2   R7,R6
        MOVF    R6,-4(R10)
        MOVF    0(R10),R6
        MOVF    #9.0,R7
        SUBF2   R7,R6
        MOVF    -4(R10),R7
        PUSHL   R6
        MOVF    #54.0,R6
        ADDF2   R7,R6
        DIVF2   R7,R6
        MOVF    #5.0,R7
        MNEGF   R7,R7
        MULF2   R7,R6
        MOVF    R6,0(R10)
        PUSHL   #4
        PUSHL   #0
        BRB     S1
S0:     .ASCII  \IIII\
S1:
        PUSHAB  S0
        PUSHL   #4
        PUSHAB  G PASSFV OUTPUT
        CALLS   #5,G PASSWRITE STRING
        PUSHAB  G PASSFV INPUT
        CALLS   #1,G PASSREAD REALF F
        MOVL    R10,R6
        ADDL2   #0,R6
        MOVF    R0,(R6)
        MOVF    0(R10),R6
```

```
        PUSHL   #5
        PUSHL   #10
        PUSHL   R6
        PUSHAB  G PASSFV OUTPUT
        CALLS   #4,G PASSWRITE REALF F
        MOVF    -4(R10),R6
        PUSHL   #5
        PUSHL   #10
        PUSHL   R6
        PUSHAB  G PASSFV OUTPUT
        CALLS   #4,G PASSWRITE REALF F
C999:
        $EXIT S
        .END    1.99
```

```
برنـــامج اوجد
<*
   متـغيـرات :  صحيح : س، ص، ع ، موقت ، قـاسم
*>

<!
   اذا ( ~  ( س< ص) )
        <: ( صواب ) : <!
              ص:= موقت !>
          <:

<!
```

```
                .ENTRY  I.99, M< >
                SUBL2   #20,SP
                MOVL    FP,R10
                MOVL    0(R10),R6
                MOVL    -4(R10),R7
                CMPL    R6,R7
                BGEQ    B0
                MOVL    #1,R9
B0:
                MOVL    R9,R6
                CLRL    R9
                TSTB    R6
                BNEQ    B1
                MOVL    #1,R9
B1:
                MOVL    R9,R6
                CLRL    R9
                MOVL    #1,R7
                TSTB    R7
                MOVL    #1,R7
                CMPB    R6,R7
                BEQL    C202
                JMP     C204
C202:
                MOVL    -12(R10),R6
                MOVL    R6,-4(R10)
C204:
C200:
C999:
                $EXIT_S
                .END    I.99
```

```
برنامج اوجد
*>
  متغيرات :  صحيح : س، ص، ع ، موقت ، قاسم
<*

!>
  اذا ((س< ه) & (س=ه) & (ه<ع) & (ع<=ه))
                    <: ( صواب ) : <!
            ص:= موقت !>
        <:

<!
```

```
                .ENTRY  I.99, M< >
                SUBL2   #20,SP
                MOVL    FP,R10
                MOVL    0(R10),R6
                MOVL    -4(R10),R7
                CMPL    R6,R7
                BGEQ    B0
                MOVL    0(R10),R6
                MOVL    -4(R10),R7
                CMPL    R6,R7
                BNEQ    B1
                MOVL    0(R10),R6
                MOVL    -4(R10),R7
                CMPL    R6,R7
                BLEQ    B2
                MOVL    0(R10),R6
                MOVL    -4(R10),R7
                CMPL    R6,R7
                BLSS    B3
                MOVL    #1,R9
B3:
B0 = B3
B1 = B0
B2 = B1
                MOVL    R9,R6
                CLRL    R9
                MOVL    #1,R7
                TSTB    R7
                MOVL    #1,R7
                CMPB    R6,R7
                BEQL    C202
                JMP     C204
C202:
                MOVL    -12(R10),R6
                MOVL    R6,-4(R10)
C204:
C200:
C999:
                $EXIT_S
                .END    I.99
```

```
برنامج !!!
<*
   متغيرات :
      صحيح: س
      *>

<#
   س:=٤
   اذا ( س= ٥ )
<! (صواب)  :  س:= س+ ١٠
   غير     :  س:= س+ ٢٠   !>
      #>
```

```
                .ENTRY  I.99, M< >
                SUBL2   #4,SP
                MOVL    FP,R10
                MOVL    #4,0(R10)
                MOVL    0(R10),R6
                MOVL    #5,R7
                CMPL    R6,R7
                BNEQ    B0
                MOVL    #1,R9
B0:
                MOVL    R9,R6
                CLRL    R9
                MOVL    #1,R7
                TSTB    R7
                MOVL    #1,R7
                CMPB    R6,R7
                BEQL    C202
                JMP     C204
C202:
                MOVL    0(R10),R6
                MOVL    #10,R7
                ADDL2   R7,R6
                MOVL    R6,0(R10)
                JMP     C200
C204:
                MOVL    0(R10),R6
                MOVL    #20,R7
                ADDL2   R7,R6
                MOVL    R6,0(R10)
C200:
C999:
                $EXIT_S
                .END    I.99
```

```
برنامج اوجد
<*
  متغيرات :  صحيح : س، ص، ع ، موقت ، قاسم
*>

<!
  اذا ((س< ص) ! (س=ص) ! (ص<س) ! (س=<ص))
        <: ( صواب ) : <!
            ص:= موقت !>
      <:

<!
```

```
                .ENTRY  I.99, M< >
                SUBL2   #20,SP
                MOVL    FP,R10
                MOVL    0(R10),R6
                MOVL    -4(R10),R7
                CMPL    R6,R7
                BLSS    B0
                MOVL    0(R10),R6
                MOVL    -4(R10),R7
                CMPL    R6,R7
                BEQL    B1
                MOVL    0(R10),R6
                MOVL    -4(R10),R7
                CMPL    R6,R7
                BGTR    B2
                MOVL    0(R10),R6
                MOVL    -4(R10),R7
                CMPL    R6,R7
                BLSS    B3
B0:
B1 = B0
B2 = B1
                MOVL    #1,R9
B3:
                MOVL    R9,R6
                CLRL    R9
                MOVL    #1,R7
                TSTB    R7
                MOVL    #1,R7
                CMPB    R6,R7
                BEQL    C202
                JMP     C204
C202:
                MOVL    -12(R10),R6
                MOVL    R6,-4(R10)
C204:
C200:
C999:
                $EXIT_S
                .END    I.99
```

```
برنامج اكبر_رقم
؟ هذا البرنامج يقرا مجموعة ارقام و يوجد اكبر رقم تم ادخاله.يتوقف
؟ البرنامج عند ادخال الرقم (-٩٩٩٩).
<* متغيرات :
        صحيح : العدد ، كبير      *>
<* كبير := صفر
   العدد := صفر
   طالما العدد >< ٩٩
<*          اذا العدد > كبير <#
        ( صواب)  : كبير := العدد    #>
        اقرا ( العدد )    *>
   اطبع (كبير)    *>
```

```
        .ENTRY  L99, M< >
        SUBL2   #8,SP
        MOVL    FP,R10
        MOVL    #0,-4(R10)
        MOVL    #0,0(R10)
C200:
        MOVL    0(R10),R6
        MOVL    #99,R7
        CMPL    R6,R7
        BEQL    C202
        MOVL    0(R10),R6
        MOVL    -4(R10),R7
        CMPL    R6,R7
        BLEQ    B0
        MOVL    #1,R9
B0:
        MOVL    R9,R6
        CLRL    R9
        MOVL    #1,R7
        TSTB    R7
        MOVL    #1,R7
        CMPB    R6,R7
        BEQL    C206
        JMP     C208
C206:
        MOVL    0(R10),R6
        MOVL    R6,-4(R10)
C208:
C204:
        PUSHAB  G PASSFV  INPUT
        CALLS   #1,G PASSREAD  INTEGER
        MOVL    R10,R6
        ADDL2   #0,R6
        MOVL    R0,(R6)
        JMP     C200
C202:
        MOVL    -4(R10),R6
        PUSHL   #10
        PUSHL   R6
        PUSHAB  G PASSFV  OUTPUT
        CALLS   #3,G PASSWRITE  INTEGER
C999:
        $EXIT_S
        .END    L99
```

```
برنامج اوجد
*>
   متغيرات : صحيح : س، ص، ع ، موقت ، قاسم
<*

!>
   اطبع ( ' ادخل الرقمين الصحيحين ' )
   اقرا ( س، ص)
   اذا (س< ص)
      <: ( صواب ) : <! موقت :=  س
                    س:= ص
                    ص:= موقت !>
      <:
   ع :=  س- ( س/ ص) * ص
   طالما ع <> .
      <*  س:= ص
          ص:= ع
          ع :=  س- ( س/ ص) * ص    *>
      قاسم :=  ص
اطبع ( ' القاسم المشترك الاعظم هو ' ، قاسم، :@ )  !>
```

```
            .ENTRY  L99, M< >
            SUBL2   #20,SP
            MOVL    FP,R10
            PUSHL   #23
            PUSHL   #0
            BRB     S1
S0:     .ASCII  \ EPOh EhRcioj EhVNoNoj \
S1:
            PUSHAB  S0
            PUSHL   #23
            PUSHAB  G PASSFV  OUTPUT
            CALLS   #5,G PASSWRITE  STRING
            PUSHAB  G PASSFV  INPUT
            CALLS   #1,G PASSREAD  INTEGER
            MOVL    R10,R6
            ADDL2   #0,R6
            MOVL    R0,(R6)
            PUSHAB  G PASSFV  INPUT
            CALLS   #1,G PASSREAD  INTEGER
            MOVL    R10,R6
            ADDL2   #-4,R6
            MOVL    R0,(R6)
            MOVL    0(R10),R6
            MOVL    -4(R10),R7
            CMPL    R6,R7
            BGEQ    B2
            MOVL    #1,R9
B2:
            MOVL    R9,R6
            CLRL    R9
            MOVL    #1,R7
            TSTB    R7
            MOVL    #1,R7
            CMPB    R6,R7
            BEQL    C202
            JMP     C204
C202:
            MOVL    0(R10),R6
            MOVL    R6,-12(R10)
            MOVL    -4(R10),R6
            MOVL    R6,0(R10)
            MOVL    -12(R10),R6
            MOVL    R6,-4(R10)
C204:
C200:
            MOVL    0(R10),R6
            MOVL    0(R10),R7
            PUSHL   R6
```

```
            MOVL    -4(R10),R6
            PUSHL   R6
            MOVL    R7,R6
            MOVL    (SP)+,R7
            DIVL2   R7,R6
            MOVL    -4(R10),R7
            MULL2   R7,R6
            PUSHL   R6
            MOVL    R7,R6
            MOVL    (SP)+,R7
            MOVL    (SP)+,R6
            SUBL2   R7,R6
            MOVL    R6,-8(R10)
C206:
            MOVL    -8(R10),R6
            MOVL    #0,R7
            CMPL    R6,R7
            BEQL    C208
            MOVL    -4(R10),R6
            MOVL    R6,0(R10)
            MOVL    -8(R10),R6
            MOVL    R6,-4(R10)
            MOVL    0(R10),R6
            MOVL    0(R10),R7
            PUSHL   R6
            MOVL    -4(R10),R6
            PUSHL   R6
            MOVL    R7,R6
            MOVL    (SP)+,R7
            DIVL2   R7,R6
            MOVL    -4(R10),R7
            MULL2   R7,R6
            PUSHL   R6
            MOVL    R7,R6
            MOVL    (SP)+,R7
            MOVL    (SP)+,R6
            SUBL2   R7,R6
            MOVL    R6,-8(R10)
            JMP     C206
C208:
            MOVL    -4(R10),R6
            MOVL    R6,-16(R10)
            PUSHL   #26
            PUSHL   #0
            BRB     S4
S3:     .ASCII  \ EhcETi EhiUJRd EfZYi km  \
S4:
            PUSHAB  S3
```

```
        PUSHL   #26
        PUSHAB  G PASSFV OUTPUT
        CALLS   #5,G PASSWRITE STRING
        MOVL    -16(R10),R6
        PUSHL   #10
        PUSHL   R6
        PUSHAB  G PASSFV OUTPUT
        CALLS   #3,G PASSWRITE INTEGER
        PUSHAB  G PASSFV OUTPUT
        CALLS   #1,G PASSWRITELN2
C999:
        $EXIT S
        .END    1.99
```

```
برنامج افرز
؟ هذا البرنامج يقوم بقراءة مصفوفه ثم يفرزها تصاعديا.
<* ثوابت :     صحيح : سعة =5
   نماذج  : ن = مصفوفة (سعة ) :صحيح
   متغيرات :  صحيح : س، موقت، طط
              منطقي : بدل
              ن : ص          *>
<!
   طط:= سعة-1
   لقيم س من 1 الى سعة بمقدار 1
      <* اقرا ( ه[س] )  *>
      بدل :=صواب
      طالما بدل
         <* بدل :=خطا
         لقيم س من 1 الى طط   بمقدار 1
            <! اذا ه[س] > ه[س+1]
               <* (صواب) : موقت :=ه[س]
               ه[س] :=ه[س+1]
               ه[س+1]:=موقت
               بدل :=صواب      *>  !>  *>
   لقيم س من 1 الى سعة بمقدار 1
      <* اطبع ( ه[س] )  *>
!>
```

```
        .ENTRY  I.99, M< >
        SUBL2   #33,SP
        MOVL    FP,R10
        MOVL    #5,R6
        MOVL    #1,R7
        SUBL2   R7,R6
        MOVL    R6,-8(R10)
        MOVL    #1,0(R10)
C200:
        MOVL    0(R10),R6
        MOVL    R6,R5
        ADDL2   R5,R5
        ADDL2   R5,R5
        MOVL    #-33,R11
        ADDL2   R5,R11
        ADDL2   R10,R11
        PUSHL   R11
        PUSHAB  G PASSFV INPUT
        CALLS   #1,G PASSREAD INTEGER
        MOVL    R0,(SP)+
        MOVL    #1,R6
        MOVL    0(R10),R7
        ADDL2   R7,R6
        MOVL    R6,0(R10)
        MOVL    0(R10),R6
        MOVL    #5,R7
        CMPL    R6,R7
        BLEQ    C200
C202:
        MOVL    #1,R6
        TSTB    R6
        MOVB    #1,-12(R10)
C204:
        MOVZBL  -12(R10),R6
        TSTB    R6
        BEQL    C206
        MOVL    #0,R6
        TSTB    R6
        MOVB    #0,-12(R10)
        MOVL    #1,0(R10)
C208:
        MOVL    0(R10),R6
        MOVL    R6,R5
        ADDL2   R5,R5
        ADDL2   R5,R5
        MOVL    #-33,R11
        ADDL2   R5,R11
        ADDL2   R10,R11
```

```
        PUSHL   R11
        MOVL    (SP)+,R11
        MOVL    (R11),R6
        MOVL    0(R10),R7
        PUSHL   R6
        MOVL    #1,R6
        ADDL2   R7,R6
        MOVL    R6,R5
        ADDL2   R5,R5
        ADDL2   R5,R5
        MOVL    #-33,R11
        ADDL2   R5,R11
        ADDL2   R10,R11
        PUSHL   R11
        MOVL    (SP)+,R11
        MOVL    (R11),R6
        MOVL    (SP)+,R7
        CMPL    R6,R7
        BGTR    B0
        MOVL    #1,R9
B0:
        MOVL    R9,R6
        CLRL    R9
        MOVL    #1,R7
        TSTB    R7
        MOVL    #1,R7
        CMPB    R6,R7
        BEQL    C214
        JMP     C216
C214:
        MOVL    0(R10),R6
        MOVL    R6,R5
        ADDL2   R5,R5
        ADDL2   R5,R5
        MOVL    #-33,R11
        ADDL2   R5,R11
        ADDL2   R10,R11
        PUSHL   R11
        MOVL    (SP)+,R11
        MOVL    (R11),R6
        MOVL    R6,-4(R10)
        MOVL    0(R10),R6
        MOVL    R6,R5
        ADDL2   R5,R5
        ADDL2   R5,R5
        MOVL    #-33,R11
        ADDL2   R5,R11
        ADDL2   R10,R11
```

```
        PUSHL   R11
        MOVL    0(R10),R6
        MOVL    #1,R7
        ADDL2   R7,R6
        MOVL    R6,R5
        ADDL2   R5,R5
        ADDL2   R5,R5
        MOVL    #-33,R11
        ADDL2   R5,R11
        ADDL2   R10,R11
        PUSHL   R11
        MOVL    (SP)+,R11
        MOVL    (R11),R6
        MOVL    (SP)+,R11
        MOVL    R6,(R11)
        MOVL    0(R10),R6
        MOVL    #1,R7
        ADDL2   R7,R6
        MOVL    R6,R5
        ADDL2   R5,R5
        ADDL2   R5,R5
        MOVL    #-33,R11
        ADDL2   R5,R11
        ADDL2   R10,R11
        PUSHL   R11
        MOVL    -4(R10),R6
        MOVL    (SP)+,R11
        MOVL    R6,(R11)
        MOVL    #1,R6
        TSTB    R6
        MOVB    #1,-12(R10)
C216:
C212:
        MOVL    #1,R6
        MOVL    0(R10),R7
        ADDL2   R7,R6
        MOVL    R6,0(R10)
        MOVL    0(R10),R6
        MOVL    -8(R10),R7
        CMPL    R6,R7
        BLEQ    C208
C210:
        JMP     C204
C206:
        MOVL    #1,0(R10)
C218:
        MOVL    0(R10),R6
        MOVL    R6,R5
```

```
        ADDL2   R5,R5
        ADDL2   R5,R5
        MOVL    #-33,R11
        ADDL2   R5,R11
        ADDL2   R10,R11
        PUSHL   R11
        MOVL    (SP)+,R11
        MOVL    (R11),R6
        PUSHL   #10
        PUSHL   R6
        PUSHAB  G PASSFV OUTPUT
        CALLS   #3,G PASSWRITE INTEGER
        MOVL    #1,R6
        MOVL    0(R10),R7
        ADDL2   R7,R6
        MOVL    R6,0(R10)
        MOVL    0(R10),R6
        MOVL    #5,R7
        CMPL    R6,R7
        BLEQ    C218
C220:
C999:
        $EXIT S
        .END    1.99
```

```
برنامج حول
<*
    نماذج: ل = مصفوفة (٥٠,١٠) : صحيح
    متغيرات:    ل: ج
    صحيح: موقت،باقي، س، ص،ن،عداد
*>
<! عداد :=  .
   س:= ١
  اطبع ( 'ادخل الرقم العشري الصحيح ' )
  اقرا( ص)
  اذا ص< .
        <:    ( صواب) :  ص:= ص* -١ :>
اعد
                ن :=  ص/٢
                اطبع(:@،ن)
                موقت :=  ن * ٢
                باقي :=  ص- موقت
                ج [س] :=  باقي
                عداد :=  عداد + ١
                س:=  س+ ١
                ص:=  ن
            حتى  ص =  .
            اطبع(:@،عداد)
اطبع ( ' الرقم الثنائي المقابل للرقم العشري هو ' )
لقيم سمن ١ الى عداد بمقدار ١
        <# اطبع ( ج [س] )  #>      !>
```

```
                .ENTRY  I.99, M< >
                SUBL2   #224,SP
                MOVL    FP,R10
                MOVL    #0,-220(R10)
                MOVL    #1,-208(R10)
                PUSHL   #26
                PUSHL   #0
                BRB     S1
S0:     .ASCII  \EPOh EhRci EhZURo  EhVNoN \
S1:
                PUSHAB  S0
                PUSHL   #26
                PUSHAB  G PASSFV  OUTPUT
                CALLS   #5,G PASSWRITE  STRING
                PUSHAB  G PASSFV  INPUT
                CALLS   #1,G PASSREAD  INTEGER
                MOVL    R10,R6
                ADDL2   #-212,R6
                MOVL    R0,(R6)
                MOVL    -212(R10),R6
                MOVL    #0,R7
                CMPL    R6,R7
                BGEQ    B2
                MOVL    #1,R9
B2:
                MOVL    R9,R6
                CLRL    R9
                MOVL    #1,R7
                TSTB    R7
                MOVL    #1,R7
                CMPB    R6,R7
                BEQL    C202
                JMP     C204
C202:
                MOVL    -212(R10),R6
                MOVL    #1,R7
                MNEGW   R7,R7
                MULL2   R7,R6
                MOVL    R6,-212(R10)
C204:
C200:
C206:
                MOVL    -212(R10),R6
                MOVL    #2,R7
                DIVL2   R7,R6
                MOVL    R6,-216(R10)
                MOVL    -216(R10),R6
                PUSHL   #10
```

```
        PUSHL   R6
        PUSHAB  G PASSFV OUTPUT
        CALLS   #3,G PASSWRITE INTEGER
        PUSHAB  G PASSFV OUTPUT
        CALLS   #1,G PASSWRITELN2
        MOVL    -216(R10),R6
        MOVL    #2,R7
        MULL2   R7,R6
        MOVL    R6,-200(R10)
        MOVL    -212(R10),R6
        MOVL    -200(R10),R7
        SUBL2   R7,R6
        MOVL    R6,-204(R10)
        MOVL    -208(R10),R6
        MOVL    R6,R5
        ADDL2   R5,R5
        ADDL2   R5,R5
        MOVL    #-200,R11
        ADDL2   R5,R11
        ADDL2   R10,R11
        PUSHL   R11
        MOVL    -204(R10),R6
        MOVL    (SP)+,R11
        MOVL    R6,(R11)
        MOVL    -220(R10),R6
        MOVL    #1,R7
        ADDL2   R7,R6
        MOVL    R6,-220(R10)
        MOVL    -208(R10),R6
        MOVL    #1,R7
        ADDL2   R7,R6
        MOVL    R6,-208(R10)
        MOVL    -216(R10),R6
        MOVL    R6,-212(R10)
        MOVL    -212(R10),R6
        MOVL    #0,R7
        CMPL    R6,R7
        BNEQ    C206
C208:
        MOVL    -220(R10),R6
        PUSHL   #10
        PUSHL   R6
        PUSHAB  G PASSFV OUTPUT
        CALLS   #3,G PASSWRITE INTEGER
        PUSHAB  G PASSFV OUTPUT
        CALLS   #1,G PASSWRITELN2
        PUSHL   #39
        PUSHL   #0
```

```
            BRB     S4
S3:     .ASCII  \ EhRci EhIjEno EhicEIh hhRci FhZURe km \
S4:
            PUSHAB  S3
            PUSHL   #39
            PUSHAB  G PASSFV  OUTPUT
            CALLS   #5,G PASSWRITE  STRING
            MOVL    #1,-208(R10)
C210:
            MOVL    -208(R10),R6
            MOVL    R6,R5
            ADDL2   R5,R5
            ADDL2   R5,R5
            MOVL    #-200,R11
            ADDL2   R5,R11
            ADDL2   R10,R11
            PUSHL   R11
            MOVL    (SP)+,R11
            MOVL    (R11),R6
            PUSHL   #10
            PUSHL   R6
            PUSHAB  G PASSFV  OUTPUT
            CALLS   #3,G PASSWRITE  INTEGER
            MOVL    #1,R6
            MOVL    -208(R10),R7
            ADDL2   R7,R6
            MOVL    R6,-208(R10)
            MOVL    -208(R10),R6
            MOVL    -220(R10),R7
            CMPL    R6,R7
            BLEQ    C210
C212:
C999:
            $EXIT_S
            .END    L99
```

```
برنامج افرز
؟ هذا البرنامج يقوم بقراءة مصفوفه ثم يفرزها تصاعديا.
<* ثوابت :     صحيح : سعة =٥
   نماذج : ن = مصفوفة (سعة،سعة،سعة ) :صحيح
      متغيرات :  صحيح : س، موقت،طط
                 منطقي : بدل
                  ن : ص          *>
<!
   طط:= سعة-١
؟  لقيم س من ١ الى سعة بمقدار ١
؟       <* اقرا ( ص[س] )  *>
                  ص[١،١،١]:=٥٠٠
                  ص[٢،٢،٢]:=٣٥٠٠
                  ص[٣،٣،٣]:=١٥٠٠
                  ص[٤،٤،٤]:=٩٥٠٠
                  ص[٥،٥،٥]:=-٥٠٠
                     بدل :=  صواب
                     طالما بدل
                  <* بدل :=  خطا
          لقيم س من ١ الى طط     بمقدار ١
        <! اذا ص[س، س، س] > ص[س+١، س+١، س+١]
                  <*  (صواب) :
                موقت :=  ص[س، س، س]
ص[س، س، س]  :=  ص[س+١، س+١، س+١]
   ص[س+١، س+١، س+١]:= موقت
                بدل :=  صواب
                                <*
                                  <!
                                    <*
          لقيم س من ١ الى سعة  بمقدار ١
          <* اطبع ( ص[س، س، س]:١٠  )  *>
                     اطبع(موقت:١٠)
                                  <!
```

```
.ENTRY  I.99, M< >
SUBL2   #513,SP
MOVL    FP,R10
MOVL    #5,R6
MOVL    #1,R7
SUBL2   R7,R6
MOVL    R6,-8(R10)
MOVL    #1,R6
MOVL    #100,R7
MULL2   R7,R6
MOVL    #1,R7
PUSHL   R6
MOVL    #20,R6
MULL2   R7,R6
MOVL    (SP)+,R7
ADDL2   R7,R6
MOVL    #1,R7
MOVL    R7,R5
ADDL2   R5,R5
ADDL2   R5,R5
ADDL2   R6,R5
MOVL    #-633,R11
ADDL2   R5,R11
ADDL2   R10,R11
PUSHL   R11
MOVL    (SP)+,R11
MOVL    #500,(R11)
MOVL    #2,R6
MOVL    #100,R7
MULL2   R7,R6
MOVL    #2,R7
PUSHL   R6
MOVL    #20,R6
MULL2   R7,R6
MOVL    (SP)+,R7
ADDL2   R7,R6
MOVL    #2,R7
MOVL    R7,R5
ADDL2   R5,R5
ADDL2   R5,R5
ADDL2   R6,R5
MOVL    #-633,R11
ADDL2   R5,R11
ADDL2   R10,R11
PUSHL   R11
MOVL    (SP)+,R11
MOVL    #3500,(R11)
MOVL    #3,R6
```

```
MOVL    #100,R7
MULL2   R7,R6
MOVL    #3,R7
PUSHL   R6
MOVL    #20,R6
MULL2   R7,R6
MOVL    (SP)+,R7
ADDL2   R7,R6
MOVL    #3,R7
MOVL    R7,R5
ADDL2   R5,R5
ADDL2   R5,R5
ADDL2   R6,R5
MOVL    #-633,R11
ADDL2   R5,R11
ADDL2   R10,R11
PUSHL   R11
MOVL    (SP)+,R11
MOVL    #1500,(R11)
MOVL    #4,R6
MOVL    #100,R7
MULL2   R7,R6
MOVL    #4,R7
PUSHL   R6
MOVL    #20,R6
MULL2   R7,R6
MOVL    (SP)+,R7
ADDL2   R7,R6
MOVL    #4,R7
MOVL    R7,R5
ADDL2   R5,R5
ADDL2   R5,R5
ADDL2   R6,R5
MOVL    #-633,R11
ADDL2   R5,R11
ADDL2   R10,R11
PUSHL   R11
MOVL    (SP)+,R11
MOVL    #9500,(R11)
MOVL    #5,R6
MOVL    #100,R7
MULL2   R7,R6
MOVL    #5,R7
PUSHL   R6
MOVL    #20,R6
MULL2   R7,R6
MOVL    (SP)+,R7
ADDL2   R7,R6
```

```
        MOVL    #5,R7
        MOVL    R7,R5
        ADDL2   R5,R5
        ADDL2   R5,R5
        ADDL2   R6,R5
        MOVL    #-633,R11
        ADDL2   R5,R11
        ADDL2   R10,R11
        PUSHL   R11
        MOVL    #500,R6
        MNEGW   R6,R6
        MOVL    (SP)+,R11
        MOVL    R6,(R11)
        MOVL    #1,R6
        TSTB    R6
        MOVB    #1,-12(R10)
C200:
        MOVZBL  -12(R10),R6
        TSTB    R6
        BEQL    C202
        MOVL    #0,R6
        TSTB    R6
        MOVB    #0,-12(R10)
        MOVL    #1,0(R10)
C204:
        MOVL    0(R10),R6
        MOVL    #100,R7
        MULL2   R7,R6
        MOVL    0(R10),R7
        PUSHL   R6
        MOVL    #20,R6
        MULL2   R7,R6
        MOVL    (SP)+,R7
        ADDL2   R7,R6
        MOVL    0(R10),R7
        MOVL    R7,R5
        ADDL2   R5,R5
        ADDL2   R5,R5
        ADDL2   R6,R5
        MOVL    #-633,R11
        ADDL2   R5,R11
        ADDL2   R10,R11
        PUSHL   R11
        MOVL    (SP)+,R11
        MOVL    (R11),R6
        MOVL    0(R10),R7
        PUSHL   R6
        MOVL    #1,R6
```

```
        ADDL2   R7,R6
        MOVL    #100,R7
        MULL2   R7,R6
        MOVL    0(R10),R7
        PUSHL   R6
        MOVL    #1,R6
        ADDL2   R7,R6
        MOVL    #20,R7
        MULL2   R7,R6
        MOVL    (SP)+,R7
        ADDL2   R7,R6
        MOVL    0(R10),R7
        PUSHL   R6
        MOVL    #1,R6
        ADDL2   R7,R6
        MOVL    R6,R5
        ADDL2   R5,R5
        ADDL2   R5,R5
        ADDL2   R6,R5
        MOVL    #-633,R11
        ADDL2   R5,R11
        ADDL2   R10,R11
        PUSHL   R11
        MOVL    (SP)+,R11
        MOVL    (R11),R6
        MOVL    (SP)+,R7
        CMPL    R6,R7
        BGTR    B0
        MOVL    #1,R9
B0:
        MOVL    R9,R6
        CLRL    R9
        MOVL    #1,R7
        TSTB    R7
        MOVL    #1,R7
        CMPB    R6,R7
        BEQL    C210
        JMP     C212
C210:
        MOVL    0(R10),R6
        MOVL    #100,R7
        MULL2   R7,R6
        MOVL    0(R10),R7
        PUSHL   R6
        MOVL    #20,R6
        MULL2   R7,R6
        MOVL    (SP)+,R7
        ADDL2   R7,R6
```

```
MOVL    0(R10),R7
MOVL    R7,R5
ADDL2   R5,R5
ADDL2   R5,R5
ADDL2   R6,R5
MOVL    #-633,R11
ADDL2   R5,R11
ADDL2   R10,R11
PUSHL   R11
MOVL    (SP)+,R11
MOVL    (R11),R6
MOVL    R6,-4(R10)
MOVL    0(R10),R6
MOVL    #100,R7
MULL2   R7,R6
MOVL    0(R10),R7
PUSHL   R6
MOVL    #20,R6
MULL2   R7,R6
MOVL    (SP)+,R7
ADDL2   R7,R6
MOVL    0(R10),R7
MOVL    R7,R5
ADDL2   R5,R5
ADDL2   R5,R5
ADDL2   R6,R5
MOVL    #-633,R11
ADDL2   R5,R11
ADDL2   R10,R11
PUSHL   R11
MOVL    0(R10),R6
MOVL    #1,R7
ADDL2   R7,R6
MOVL    #100,R7
MULL2   R7,R6
MOVL    0(R10),R7
PUSHL   R6
MOVL    #1,R6
ADDL2   R7,R6
MOVL    #20,R7
MULL2   R7,R6
MOVL    (SP)+,R7
ADDL2   R7,R6
MOVL    0(R10),R7
PUSHL   R6
MOVL    #1,R6
ADDL2   R7,R6
MOVL    R6,R5
```

```
        ADDL2   R5,R5
        ADDL2   R5,R5
        ADDL2   R6,R5
        MOVL    #-633,R11
        ADDL2   R5,R11
        ADDL2   R10,R11
        PUSHL   R11
        MOVL    (SP)+,R11
        MOVL    (R11),R6
        MOVL    (SP)+,R11
        MOVL    R6,(R11)
        MOVL    0(R10),R6
        MOVL    #1,R7
        ADDL2   R7,R6
        MOVL    #100,R7
        MULL2   R7,R6
        MOVL    0(R10),R7
        PUSHL   R6
        MOVL    #1,R6
        ADDL2   R7,R6
        MOVL    #20,R7
        MULL2   R7,R6
        MOVL    (SP)+,R7
        ADDL2   R7,R6
        MOVL    0(R10),R7
        PUSHL   R6
        MOVL    #1,R6
        ADDL2   R7,R6
        MOVL    R6,R5
        ADDL2   R5,R5
        ADDL2   R5,R5
        ADDL2   R6,R5
        MOVL    #-633,R11
        ADDL2   R5,R11
        ADDL2   R10,R11
        PUSHL   R11
        MOVL    -4(R10),R6
        MOVL    (SP)+,R11
        MOVL    R6,(R11)
        MOVL    #1,R6
        TSTB    R6
        MOVB    #1,-12(R10)
C212:
C208:
        MOVL    #1,R6
        MOVL    0(R10),R7
        ADDL2   R7,R6
        MOVL    R6,0(R10)
```

```
        MOVL    0(R10),R6
        MOVL    -8(R10),R7
        CMPL    R6,R7
        BLEQ    C204
C206:
        JMP     C200
C202:
        MOVL    #1,0(R10)
C214:
        MOVL    0(R10),R6
        MOVL    #100,R7
        MULL2   R7,R6
        MOVL    0(R10),R7
        PUSHL   R6
        MOVL    #20,R6
        MULL2   R7,R6
        MOVL    (SP)+,R7
        ADDL2   R7,R6
        MOVL    0(R10),R7
        MOVL    R7,R5
        ADDL2   R5,R5
        ADDL2   R5,R5
        ADDL2   R6,R5
        MOVL    #-633,R11
        ADDL2   R5,R11
        ADDL2   R10,R11
        PUSHL   R11
        MOVL    (SP)+,R11
        MOVL    (R11),R6
        PUSHL   #10
        PUSHL   R6
        PUSHAB  G PASSFV  OUTPUT
        CALLS   #3,G PASSWRITE  INTEGER
        MOVL    #1,R6
        MOVL    0(R10),R7
        ADDL2   R7,R6
        MOVL    R6,0(R10)
        MOVL    0(R10),R6
        MOVL    #5,R7
        CMPL    R6,R7
        BLEQ    C214
C216:
        MOVL    -4(R10),R6
        PUSHL   #10
        PUSHL   R6
        PUSHAB  G PASSFV  OUTPUT
        CALLS   #3,G PASSWRITE  INTEGER
C999:
```

```
$EXIT S
.END    1.99
```

```
برنامج ٢١
<* نماذج :
ط = سلسلة
خ = مؤشر : ط
ض=ملف متسلسل: خ
متغيرات :  صحيح: ص
خ : س،ز
ض : ف
<*
<#
جهرط (ف، 'ن')
استدع جديد(س)
استدع جديد(ز)
لقيم ص=ن ١ إلى ٣ بمقدار ١
<! اطبع [ف] ( ز، س، :@)
اطبع ( :@، ص، ز، س)
ز :=  س !>
<#
```

```
                .ENTRY  I.99, M< >
                SUBL2   #312,SP
                MOVL    FP,R10
                PUSHL   #4
                BRB     S1
S0:     .ASCII  \BB\
S1:
                PUSHAB  S0
                MOVL    R10,R6
                ADDL2   #-308,R6
                MOVL    R6,R0
                MOVAB   28(R0),(R0)+
                CLRL    (R0)+
                MOVAB   8(R0),(R0)+
                CLRL    (R0)+
                CLRL    (R0)+
                MOVL    #2,(R0)+
                MOVL    #4,(R0)
                PUSHL   #2
                PUSHL   #1
                PUSHL   R6
                CALLS   #5,G PASSOPEN2
                PUSHL   R6
                CALLS   #1,G PASSREWRITE2
                MOVL    R10,R6
                ADDL2   #-4,R6
                PUSHL   R6
                MOVL    (SP)+,R6
                PUSHL   #81
                CALLS   #1,G PASSNEW2
                MOVL    R0,(R6)
                MOVL    R10,R6
                ADDL2   #-8,R6
                PUSHL   R6
                MOVL    (SP)+,R6
                PUSHL   #81
                CALLS   #1,G PASSNEW2
                MOVL    R0,(R6)
                MOVL    #1,0(R10)
C200:
                MOVL    R10,R6
                ADDL2   #-308,R6
                MOVL    R6,R8
                MOVL    -8(R10),R6
                MOVL    R6,28(R8)
                PUSHL   R8
                CALLS   #1,G PASSPUT
                MOVL    -4(R10),R6
```

```
        MOVL    R6,28(R8)
        PUSHL   R8
        CALLS   #1,G PASSPUT
        MOVL    0(R10),R6
        PUSHL   #10
        PUSHL   R6
        PUSHAB  G PASSFV OUTPUT
        CALLS   #3,G PASSWRITE INTEGER
        MOVL    -8(R10),R6
        PUSHL   #10
        PUSHL   R6
        PUSHAB  G PASSFV OUTPUT
        CALLS   #3,G PASSWRITE INTEGER
        MOVL    -4(R10),R6
        PUSHL   #10
        PUSHL   R6
        PUSHAB  G PASSFV OUTPUT
        CALLS   #3,G PASSWRITE INTEGER
        PUSHAB  G PASSFV OUTPUT
        CALLS   #1,G PASSWRITELN2
        MOVL    -4(R10),R6
        MOVL    R6,-8(R10)
        MOVL    #1,R6
        MOVL    0(R10),R7
        ADDL2   R7,R6
        MOVL    R6,0(R10)
        MOVL    0(R10),R6
        MOVL    #3,R7
        CMPL    R6,R7
        BLEQ    C200
C202:
C999:
        $EXIT_S
        .END    L99
```

برنامج ٢١

```
<* نماذج :
ف=ملف متسلسل: حقيقي
متغيرات :    صحيح: ص
          حقيقي  : س
          ض  :    ف

<*
#>
          جهرط (ف، 'ن')
          س:= ٠,٠
لقيم ص=١ الى ١٠   بمقدار ١
     <!  اطبع [ف]  (      س)
         اطبع       (      س)
     س:= س+ ٠,٢ !>

<#
```

```
            .ENTRY  I.99, M< >
            SUBL2   #308,SP
            MOVL    FP,R10
            PUSHL   #4
            BRB     S1
S0:     .ASCII  \BB\
S1:
            PUSHAB  S0
            MOVL    R10,R6
            ADDL2   #-304,R6
            MOVL    R6,R0
            MOVAB   28(R0),(R0)+
            CLRL    (R0)+
            MOVAB   8(R0),(R0)+
            CLRL    (R0)+
            CLRL    (R0)+
            MOVL    #2,(R0)+
            MOVL    #4,(R0)
            PUSHL   #2
            PUSHL   #1
            PUSHL   R6
            CALLS   #5,G PASSOPEN2
            PUSHL   R6
            CALLS   #1,G PASSREWRITE2
            MOVF    #0.0,-4(R10)
            MOVL    #1,0(R10)
C200:
            MOVL    R10,R6
            ADDL2   #-304,R6
            MOVL    R6,R8
            MOVF    -4(R10),R6
            MOVL    R6,28(R8)
            PUSHL   R8
            CALLS   #1,G PASSPUT
            MOVF    -4(R10),R6
            PUSHL   #5
            PUSHL   #10
            PUSHL   R6
            PUSHAB  G PASSFV OUTPUT
            CALLS   #4,G PASSWRITE REALF F
            MOVF    -4(R10),R6
            MOVF    #2.0,R7
            ADDF2   R7,R6
            MOVF    R6,-4(R10)
            MOVL    #1,R6
            MOVL    0(R10),R7
            ADDL2   R7,R6
            MOVL    R6,0(R10)
```

```
            MOVL    0(R10),R6
            MOVL    #10,R7
            CMPL    R6,R7
            BLEQ    C200
C202:
C999:
            $EXIT_S
            .END    L99
```

```
برنامج ا5
<* متغيرات : صحيح : ا،ب
برامج_فرعية : برنامج_فرعي مضروب( س، ط)
    متغيرات : صحيح: ط
ثوابت : صحيح: س    *>
<!   ا :=  ٧
    استدع مضروب(ا،ب)
    اطبع (ا،ب)                    !>
?
?
برنامج_فرعي مضروب( س،ط)
<* متغيرات : صحيح: ص  *>
<!   ط :=  ا
    لقيم ص من ا الى س بمقدار ا
    <# ط :=  ط  *  ص   #>      !>
```

```
        .ENTRY  L99, M< >
        SUBL2   #8,SP
        MOVL    FP,R10
        MOVL    #7,0(R10)
        MOVL    0(R10),R6
        PUSHL   R6
        MOVL    R10,R6
        ADDL2   #-4,R6
        PUSHL   R6
        PUSHL   FP
        CALLS   #2,L200
        ADDL2   #4,SP
        MOVL    0(R10),R6
        PUSHL   #10
        PUSHL   R6
        PUSHAB  G PASSFV OUTPUT
        CALLS   #3,G PASSWRITE INTEGER
        MOVL    -4(R10),R6
        PUSHL   #10
        PUSHL   R6
        PUSHAB  G PASSFV OUTPUT
        CALLS   #3,G PASSWRITE INTEGER
        JMP     C999
        .ENTRY  L200, M< >
        SUBL2   #4,SP
        MOVL    #1,R6
        MOVL    R6,@8(AP)
        MOVL    #1,0(FP)
C202:
        MOVL    @8(AP),R6
        MOVL    0(FP),R7
        MULL2   R7,R6
        MOVL    R6,@8(AP)
        MOVL    #1,R6
        MOVL    0(FP),R7
        ADDL2   R7,R6
        MOVL    R6,0(FP)
        MOVL    0(FP),R6
        MOVL    12(AP),R7
        CMPL    R6,R7
        BLEQ    C202
C204:
        RET
C999:
        $EXIT S
        .END    L99
```

```
برنامج ١-
<*متغيرات :
صحيح : ا
برامج_فرعية :
برنامج_فرعي ذ
عامة : ا
؟ برنامج_فرعي د : خارجي
<*
؟<*  *>
*>
استدع ذ
<*
برنامج_فرعي ذ
*>
متغيرات :
صحيح : أ
برامج_فرعية :
برنامج_فرعي ذ١
عامة : أ
<*
*>
ا:=٣
استدع ذ١
اطبع('لالا ',ا:١٠،' ؟؟ ',أ:١٠)
<*
برنامج_فرعي ذ١
*>
متغيرات :
صحيح : T
<*
*>
أ:= ٥
T:=أ
اطبع(' لا  ',أ:١٠،' لل ',T:١٠،@:)
<*
```

```
            .ENTRY  L99, M< >
            SUBL2   #4,SP
            MOVL    FP,R10
            PUSHL   FP
            CALLS   #0,L200
            ADDL2   #4,SP
            JMP     C999
            .ENTRY  L200, M< >
            SUBL2   #4,SP
            MOVL    #3,0(R10)
            PUSHL   FP
            CALLS   #0,L202
            ADDL2   #4,SP
            PUSHL   #3
            PUSHL   #0
            BRB     S1
S0:     .ASCII  \ee \
S1:
            PUSHAB  S0
            PUSHL   #3
            PUSHAB  G PASSFV  OUTPUT
            CALLS   #5,G PASSWRITE  STRING
            MOVL    0(R10),R6
            PUSHL   #10
            PUSHL   R6
            PUSHAB  G PASSFV  OUTPUT
            CALLS   #3,G PASSWRITE  INTEGER
            PUSHL   #4
            PUSHL   #0
            BRB     S3
S2:     .ASCII  \ FF \
S3:
            PUSHAB  S2
            PUSHL   #4
            PUSHAB  G PASSFV  OUTPUT
            CALLS   #5,G PASSWRITE  STRING
            MOVL    0(FP),R6
            PUSHL   #10
            PUSHL   R6
            PUSHAB  G PASSFV  OUTPUT
            CALLS   #3,G PASSWRITE  INTEGER
            RET
            .ENTRY  L202, M< >
            SUBL2   #4,SP
            MOVL    24(FP),R1
            MOVL    #5,0(R1)
            MOVL    24(FP),R1
            MOVL    0(R1),R6
```

```
            MOVL    R6,0(FP)
            PUSHL   #4
            PUSHL   #0
            BRB     S5
S4:     .ASCII  \ f  \
S5:
            PUSHAB  S4
            PUSHL   #4
            PUSHAB  G PASSFV  OUTPUT
            CALLS   #5,G PASSWRITE  STRING
            MOVL    24(FP),R1
            MOVL    0(R1),R6
            PUSHL   #10
            PUSHL   R6
            PUSHAB  G PASSFV  OUTPUT
            CALLS   #3,G PASSWRITE  INTEGER
            PUSHL   #4
            PUSHL   #0
            BRB     S7
S6:     .ASCII  \ hh \
S7:
            PUSHAB  S6
            PUSHL   #4
            PUSHAB  G PASSFV  OUTPUT
            CALLS   #5,G PASSWRITE  STRING
            MOVL    0(FP),R6
            PUSHL   #10
            PUSHL   R6
            PUSHAB  G PASSFV  OUTPUT
            CALLS   #3,G PASSWRITE  INTEGER
            PUSHAB  G PASSFV  OUTPUT
            CALLS   #1,G PASSWRITELN2
            RET
C999:
            $EXIT_S
            .END    I.99
```